## Copyright Notice

# Welcome!

Welcome to *The Windows Interface Guidelines for Software Design*, an indispensable guide to designing software that runs with the Microsoft® Windows® operating system. The design of your software's interface, more than anything else, affects how a user experiences your product. This guide promotes good interface design and visual and functional consistency within and across Windows-based applications.

## What's New

Continuing the direction set by Microsoft OLE, the enhancements in the Windows user interface provide a design evolution from the basic and graphical to the more object oriented — that is, from an application-centered interface to a more data-centered one. In response, developers and designers may need to rethink the interface of their software — the basic components and the respective operations and properties that apply to them. This is important because, from a user's perspective, applications have become less the primary focus and more the engines behind the objects in the interface. Users can now interact with data without having to think about applications, allowing them to better concentrate on their tasks.

When adapting your existing Windows-based software, make certain you consider the following important design aspects:

- Title bar text and icons
- Property sheets
- Transfer model (including drag and drop)
- Pop-up menus
- New controls
- Integration with the system
- Help interface
- OLE embedding and OLE linking
- Visual design of windows, controls, and icons
- Window management
- Presentation of minimized windows

These elements are covered in depth throughout this guide.

## How to Use This Guide

This guide is intended for those who are designing and developing Windows-based software. It may also be appropriate for those interested in a better understanding of the Windows environment and the human-computer interface principles it supports. The content of the guide covers the following areas:

- Basic design principles and process — fundamental design philosophy, assumptions about human behavior, design methodology, and concepts embodied in the interface.
- Interface elements — descriptive information about the various components in the interface as well as when and how to use them.
- Design details — specific information about the details of effective design and style when using the elements of the interface.
- Additional information — summary and quick reference information, a bibliography, a comprehensive word list in numerous languages to assist in product localization, and a glossary.

This guide focuses on the design and elements of an application's user interface. Although an occasional technical reference is included, this guide does not generally cover detailed information about technical implementation or application programming interfaces (APIs), because there are many different types of development tools that you can use to develop software for Windows. The documentation included with the Microsoft® Win32® Software Development Kit (SDK) is one source of information about specific APIs.

## How to Apply the Guidelines

This guide promotes visual and functional consistency within and across the Windows operating system. Although following these guidelines is encouraged, you are free to adopt the guidelines that best suit your software. However, by following these guidelines, you enable users to transfer their skills and experience from one task to the next and to learn new tasks easily. In addition, evolution toward data-centered design breaks down the lines between traditional application domains, making inconsistencies in the interface more obvious and distracting to users.

Conversely, adhering to the design guidelines does not guarantee usability. The guidelines are valuable tools, but they must be combined with other factors as part of an effective software design process, such as application of design principles, task analysis, prototyping, and usability evaluation.

You may extend these guidelines, provided that you do so in the spirit of the principles on which they are based, and maintain a reasonable level of consistency with the visual and behavioral aspects of the Windows interface. In general, avoid adding new elements or behaviors unless the interface does not otherwise support them. More importantly, avoid changing an existing behavior for common elements. A user builds up expectations about the workings of an interface. Inconsistencies not only confuse the user, they also add unnecessary complexity.

These guidelines supersede those issued for Windows version 3.1 and all previous releases and are specific to the development of applications designed for Microsoft® Windows®, Microsoft® Windows NT™ Workstation, and Microsoft® Windows NT Server. There is no direct relationship between these guidelines and those provided for other operating systems.

For more information about special considerations concerning developing applications for both Windows 95 and Windows NT operating system, see Supporting Specific Versions of Windows.

## Conventions Used in This Guide

The following conventions are used throughout this guide.

| Convention | Indicates |
|---|---|
|  | A reference to related topics in this guide or other books that provide more information about the topic. |
|  | Additional or special information about the topic. |
| SMALL CAPITAL LETTERS | Names of keys on the keyboard — for example, SHIFT, CTRL, or ALT. |
| KEY+KEY | Key combinations for which the user must press and hold down one key and then press another — for example, CTRL+P or ALT+F4. |
| *Italic text* | New terms and variable expressions, such as parameters. |
| **Bold text** | Win32 API keywords and registry key entries. |
| Registry text | Examples of registry entries. |
| [ ] | Optional information. |

# Introduction

Microsoft Windows supports the evolution and design of software from a basic graphical user interface to a data-centered interface that is better focused on users and their tasks. This topic outlines the fundamental concepts of data-centered design. It covers some of the basic definitions used throughout this guide and provides the fundamental model for how to define your interface to fit well within the Windows environment.

## Data-Centered Design

*Data-centered design* means that the design of the interface supports a model where a user can browse for data and edit it directly instead of having to first locate an appropriate editor or application. As a user interacts with data, the corresponding commands and tools to manipulate the data or the view of the data become available to the user automatically. This frees a user to focus on the information and tasks rather than on applications and how applications interact.

In this data-centered context, a *document* is a common unit of data used in tasks and exchanged between users. The use of the term is not limited to the output of a word-processing or spreadsheet application, but it emphasizes that the focus of design is on data, rather than the underlying application.

## Objects as Metaphor

A well-designed user interface provides an understandable, consistent framework in which users can work, without being confounded by the details of the underlying technology. To help accomplish this, the design model of the Windows user interface uses the metaphor of objects. This is a natural way we interpret and interact with the world around us. In the interface, *objects* not only describe files or icons, but any unit of information, including cells, paragraphs, characters, and circles, and the documents in which they reside.

## Object Characteristics

Objects, whether real-world or computer representations, have certain characteristics that help us understand what they are and how they behave. The following concepts describe the aspects and characteristics of computer representations:

- Properties — Objects have certain characteristics or attributes, called *properties*, that define their appearance or state — for example, color, size, and modification date. Properties are not limited to the external or visible traits of an object. They may reflect the internal or operational state of an object, such as an option in a spelling check utility that automatically suggests alternative spellings.
- Operations — Things that can be done with or to an object are considered its *operations*. Moving or copying an object are examples of operations. You can expose operations in the interface through a variety of mechanisms, including commands and direct manipulation.
- Relationships — Objects always exist within the context of other objects. The context, or *relationships*, that an object may have often affects the way the object appears or behaves. Common kinds of relationships include collections, constraints, and composites.

## Relationships

The simplest relationship is a *collection*, in which objects in a set share a common aspect. The results of a query or a multiple selection of objects are examples of a collection. The significance of a collection is that it enables operations to be applied to a set of objects.

A *constraint* is a stronger relationship between a set of objects in that changing an object in the set affects some other object in the set. The way a text box streams text, the way a drawing application layers its objects, and even the way a word-processing application organizes a document into pages are all examples of constraints.

When a relationship between objects becomes so significant that the aggregation can be identified as an object itself with its own set of properties and operations, the relationship is called a *composite*. A range of cells, a paragraph, and a grouped set of drawing objects are examples of composites.

Another common kind of relationship found in the interface is containment. A *container* is an object that is the place where other objects exist, such as text in a document or documents in a folder. A container often influences the behavior of its content. It may add or suppress certain properties or operations of an object placed in it. In addition, a container controls access to its content as well as what kind of object it will accept as its content. This may affect the results when transferring objects from one container to another.

All these aspects contribute to an object's *type*, a descriptive way of distinguishing or classifying objects. Objects of a common type have similar traits and behaviors.

## Composition

As in the natural world, the metaphor of objects implies a constructed environment. Objects are compositions of other objects. You can define most tasks supported by applications as a specialized combination or set of relationships between objects. A text document is a composition of text, paragraphs, footnotes, or other items. A table is a combination of cells; a chart is a particular organization of graphics. When you define user interaction with objects to be as consistent as possible at any level, you can produce complex constructions while maintaining a small, basic set of conventions. These conventions can apply throughout the interface, increasing ease of use. In addition, using composition to model tasks encourages modular, component-oriented design. This allows objects to be adapted or recombined for other uses.

## Persistence

In the natural world, objects persist in their existing state unless changed or destroyed. When you use a pen to write a note, you need not invoke a command to ensure that the ink is preserved on the paper. The act of writing implicitly preserves the information. This is the long term direction for objects in the interface as well. Although it is still appropriate to design software that requires explicit user actions to preserve data, consider whether data can be preserved automatically. In addition, view state information, such as cursor position, scroll position, and window size and location, should be preserved so it can be restored when an object's view is reopened.

## Putting Theory into Practice

Using objects in an interface design does not guarantee usability. But applying object-based concepts does offer greater potential for a well-designed interface. As with any good user interface design, a good user-centered design process ensures the success and quality of the interface.

The first step to object-based design should begin as any good design with a thorough understanding of what users' objectives and tasks are. When doing the task analysis, identify the basic components or objects used in those tasks and the behavior and the characteristics that differentiate each kind of object, including the relationships of the objects to each other and to the user. Also identify the actions that are performed, the objects to which they apply, and the state information or attributes that each object in the task must preserve, display, and allow to be edited.

Once the analysis is complete, you can start identifying the user interfaces for the objects. Define how the objects you identified are to be presented, either as icons or data elements in a form. Use icons primarily for representing composite or container objects that need to be opened into their own windows. Attribute or state information should typically be presented as properties of the associated object, most often using property sheets. Map behaviors and operations to specific kinds of interaction, such as menu commands, direct manipulation, or both. Make these accessible when the object is selected by the user. The information in this guide will help you define how to apply the interfaces provided by the system.

Redesigning an existing Windows 3.1-based application to a more data-centered interface need not require an immediate, complete overhaul. You can begin the evolution by adding contextual interfaces such as pop-up menus, property sheets, and OLE drag and drop and by following the recommendations for designing your window title bars and icons.

# Introduction

This topic provides a brief overview of some of the basic elements included in the Microsoft Windows operating system that allow the user to control the environment (sometimes collectively referred to as the *shell*). These elements provide not only the backdrop for a user's environment, but can be landmarks for the user's interaction with your application as well.

## The Desktop

The *desktop* represents a user's primary work area; it fills the screen and forms the visual background for all operations (as shown in Figure 3.1). However, the desktop is more than just a background. It can also be used as a convenient location to place objects that are stored in the file system. In addition, for a computer connected to a network, the desktop also serves as a private work area through which a user can still browse and access objects remotely located on the network.

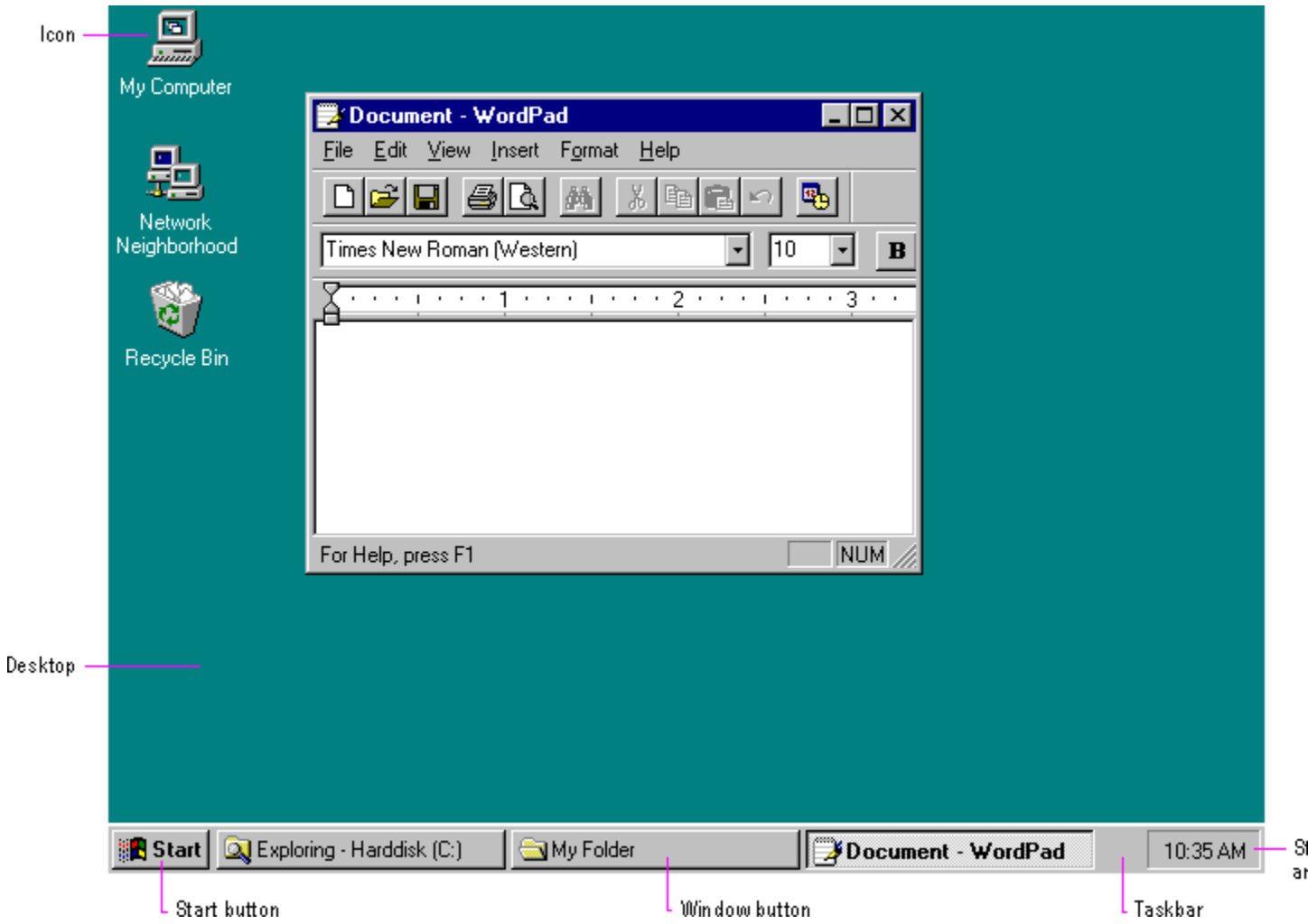



**Figure 3.1 The desktop**

## The Taskbar

The taskbar is a special component of the desktop that can be used to switch between open windows and to access global commands and other frequently used objects. As a result, it provides a home base — an operational anchor for the interface.

Like most toolbars, the taskbar can be configured. For example, a user can move the taskbar from its default location and relocate it along another edge of the screen (as shown in Figure 3.2). The user can also configure display options of the taskbar. The taskbar can provide the user access to your application. It can also be used to provide status information even when your application is not active. Because the taskbar is an interface shared across applications, be sure to follow the conventions and guidelines covered in this guide.

For more information about integrating your application with the taskbar, see Integrating with the System.

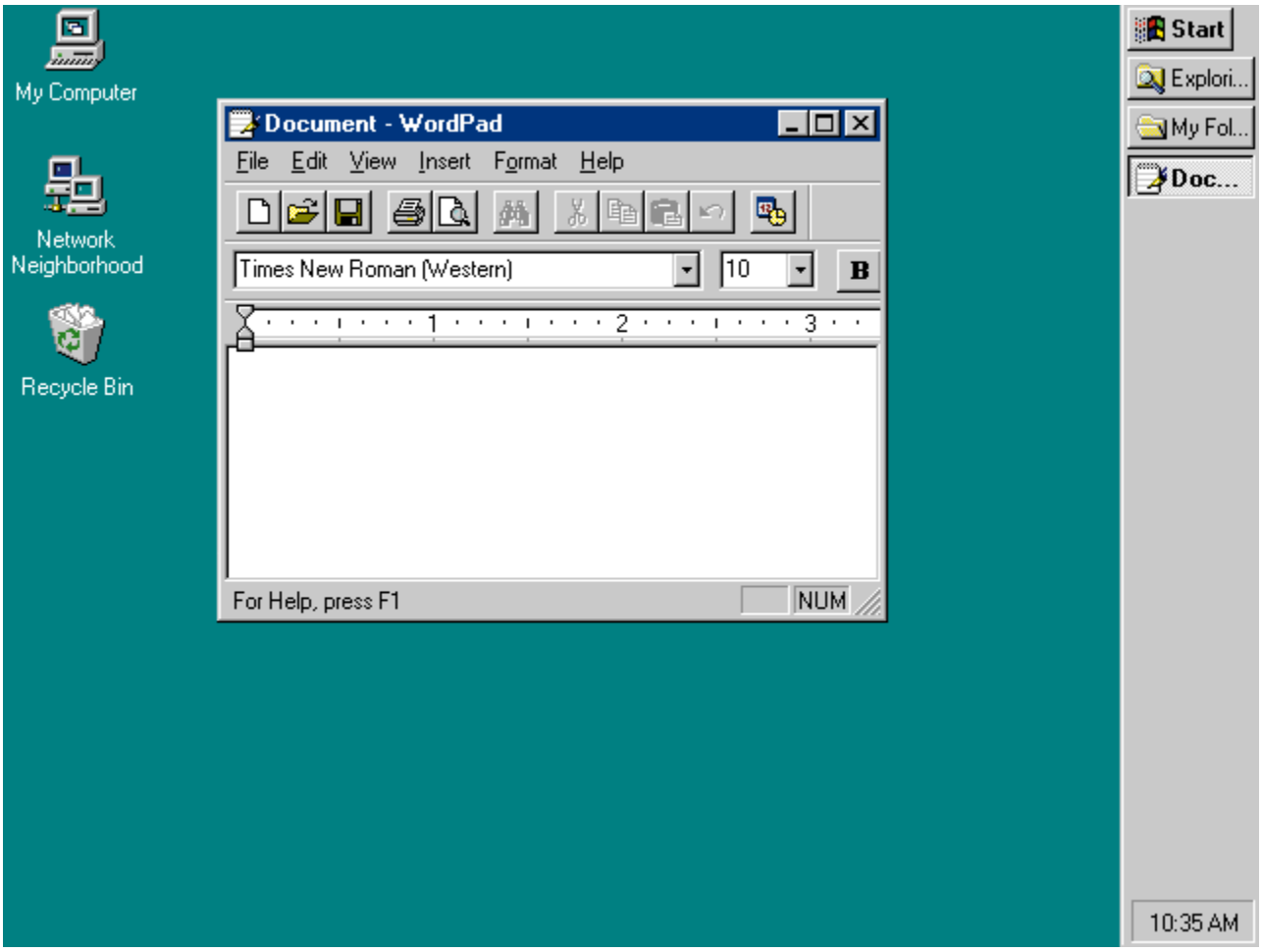



**Fig 3.2 Showing the taskbar in another location**

## The Start Button

The Start button at the left side of the taskbar displays a special menu that includes commands for opening or finding files. The Program menu entry automatically includes the Program Manager entries when the system is installed over Windows 3.1. When installing your Windows-based application, you also can include an entry for your application by placing a shortcut icon in the system's Programs folder.

## Window Buttons

Whenever the user opens a primary window, a button is placed on the taskbar for that window. This button provides the user access to the commands of that window and a convenient interface for switching to that window. The taskbar automatically adjusts the size of the buttons to accommodate as many buttons as possible. When the size of the button requires that the window's title be abbreviated, the taskbar also automatically supplies a small pop-up window (as shown in Figure 3.3) that displays the full title for the window.

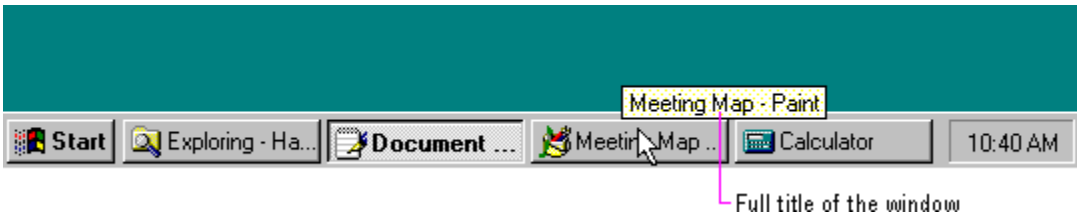



**Figure 3.3 Pop-up window with full title**

When a window is minimized, the window's button remains on the taskbar, but is removed when the window is closed.

Taskbar buttons can also be used as drag and drop destinations. When the user drags over a taskbar button, the system activates the associated window, allowing the user to drop within that window.

For more information about drag and drop, see General Interaction Techniques.

## The Status Area

On the opposite side of the taskbar from the Start menu is a special status area. Your application can place special status or notification indicators here, even when it is not active.

## Icons

Icons may appear on the desktop and in windows. *Icons* are pictorial representations of objects. This goes beyond the use of icons in Windows 3.1, which only represented minimized windows. Your software should provide and register icons for its application file and any of its associated document or data files.

For more information about the use of icons, see Integrating with the System. For information about icon design, see Visual Design.

Windows includes a number of icons that represent basic objects, such as the following.

**Table 3.1 Icons**

| Icon | Type | Function |
|---|---|---|
|  My Computer | System Folder | Provides access to a user's private storage. |
|  Network Neighborhood | System Folder | Provides access to the network. |
|  Folder | Folder | Provides organization of files and folders. |
|  Shortcut to My Favorite Folder | Shortcut | Provides access to other objects. A shortcut icon uses the icon of the type of file it is linked to, overlaid with the link symbol. |
|  All Files | Saved Search | Locates files or folders. |
|  Windows Explorer | Application | Allows browsing of the content of a user's computer or the network. |
|  Recycle Bin | System Folder | Stores deleted icons. |
|  Control Panel | System Folder | Provides access to properties of installed devices and resources (for example, fonts, displays, and keyboards). |

## Windows

You can open icons into windows. Windows provides a means of viewing and editing information, and viewing the content and properties of objects. You can also use windows to display parameters to complete commands, palettes of controls, or messages informing a user of a particular situation. Figure 3.4 demonstrates some of the different uses for windows.

For more information about windows, see Windows, and Secondary Windows.

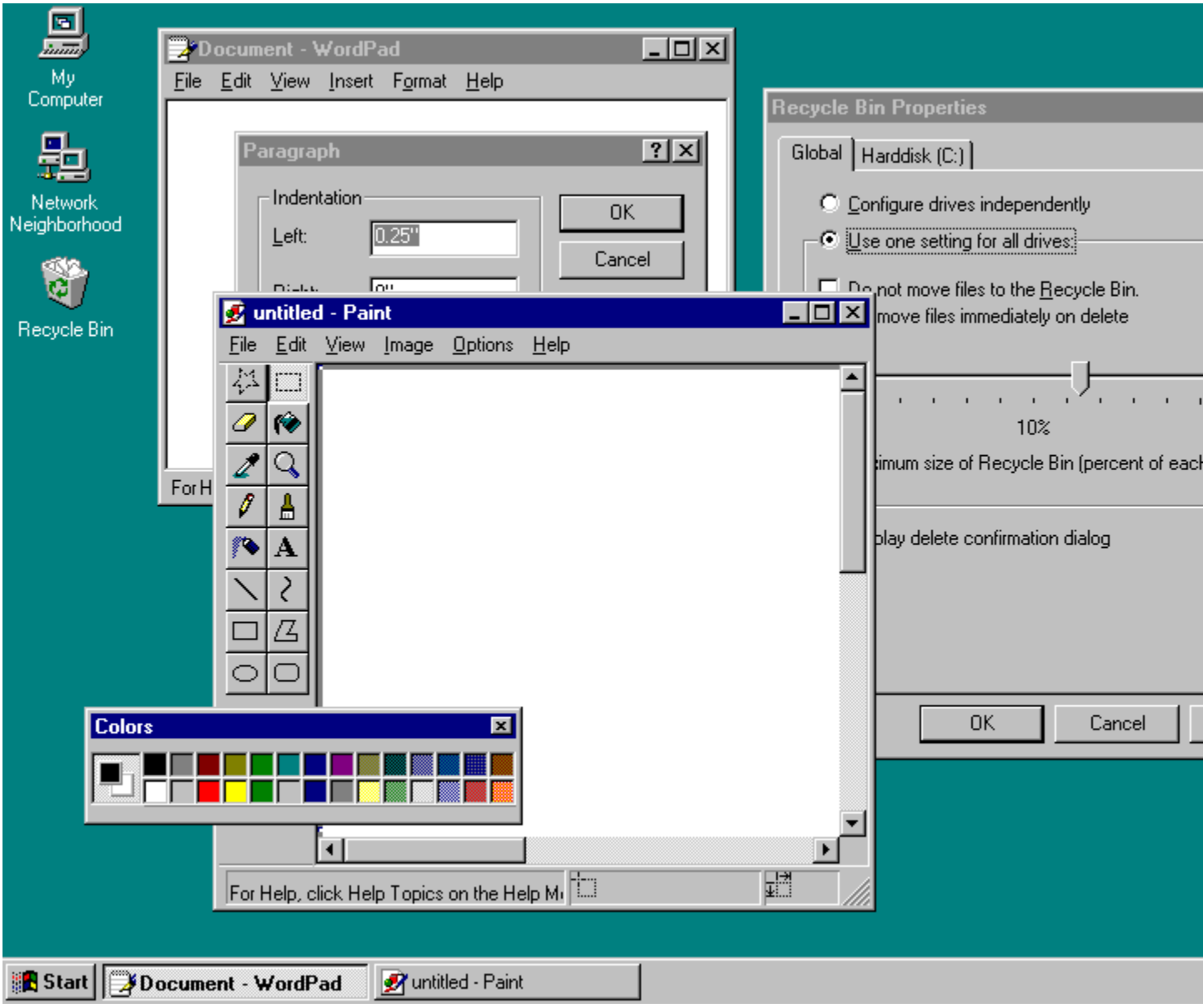



**Figure 3.4 Different uses of windows**

# Introduction

A well-designed user interface is built on principles and a development process that centers on users and their tasks. This topic summarizes the basic principles of the interface design for Microsoft Windows. It also includes techniques and methodologies employed in an effective human-computer interface design process.

## User-Centered Design Principles

The information in this section describes the design principles on which Windows and the guidelines in this book are based. You will find these principles valuable when designing software for Windows.

## User in Control

An important principle of user interface design is that the user should always feel in control of the software, rather than feeling controlled by the software. This principle has a number of implications.

The first implication is the operational assumption that the user initiates actions, not the computer or software — the user plays an active, rather than reactive, role. You can use techniques to automate tasks, but implement them in a way that allows the user to chose or control the automation.

The second implication is that users, because of their widely varying skills and preferences, must be able to personalize aspects of the interface. The system software provides user access to many of these aspects. Your software should reflect user settings for different system properties, such as color, fonts, or other options.

The final implication is that your software should be as interactive and responsive as possible. Avoid modes whenever possible. A *mode* is a state that excludes general interaction or otherwise limits the user to specific interactions. When a mode is the only or the best design alternative — for example, for selecting a particular tool in a drawing program — make certain the mode is obvious, visible, the result of an explicit user choice, and easy to cancel.

For information about applying the design principle of user in control, see Input Basics, and General Interaction Techniques. These topics cover the basic forms of interaction your software should support.

## Directness

Design your software so that users can directly manipulate software representations of information. Whether dragging an object to relocate it or navigating to a location in a document, users should see how the actions they take affect the objects on the screen. Visibility of information and choices also reduce the user's mental workload. Users can recognize a command easier than they can recall its syntax.

Familiar metaphors provide a direct and intuitive interface to user tasks. By allowing users to transfer their knowledge and experience, metaphors make it easier to predict and learn the behaviors of software-based representations.

When using metaphors, you need not limit a computer-based implementation to its "real world" counterpart. For example, unlike its paper-based counterpart, a folder on the Windows desktop can be used to organize a variety of objects such as printers, calculators, and other folders. Similarly, a Windows folder can be more easily resorted. The purpose of using metaphor in the interface is to provide a cognitive bridge; the metaphor is not an end in itself.

Metaphors support user recognition rather than recollection. Users remember a meaning associated with a familiar object more easily than they remember the name of a particular command.

For information about applying the principle of directness and metaphor, see General Interaction Techniques, and Visual Design. These topics cover, respectively, the use of directness in the interface (including drag and drop) and the use of metaphors when designing icons or other graphical elements.

## Consistency

Consistency allows users to transfer existing knowledge to new tasks, learn new things more quickly, and focus more on tasks because they need not spend time trying to remember the differences in interaction. By providing a sense of stability, consistency makes the interface familiar and predictable.

Consistency is important through all aspects of the interface, including names of commands, visual presentation of information, and operational behavior. To design consistency into software, you must consider several aspects:

- Consistency within a product. Present common functions using a consistent set of commands and interfaces. For example, avoid implementing a Copy command that immediately carries out an operation in one situation but in another presents a dialog box that requires a user to type in a destination. As a corollary to this example, use the same command to carry out functions that seem similar to the user.
- Consistency within the operating environment. By maintaining a high level of consistency between the interaction and interface conventions provided by Windows, your software benefits from users' ability to apply interaction skills they have already learned.
- Consistency with metaphors. If a particular behavior is more characteristic of a different object than its metaphor implies, the user may have difficulty learning to associate that behavior with an object. For example, an incinerator communicates a different model than a wastebasket for the recoverability of objects placed in it.

Although applying the principle of consistency is the primary goal of this guide, the following topics focus on the elements common to all Windows-based software: Windows, Menus, Controls, and Toolbars, and Secondary Windows. For information about closely integrating your software with the Windows environment, see Integrating with the System, and Working with OLE Embedded and OLE Linked Objects.

## Forgiveness

Users like to explore an interface and often learn by trial and error. An effective interface allows for interactive discovery. It provides only appropriate sets of choices and warns users about potential situations where they may damage the system or data, or better, makes actions reversible or recoverable.

Even within the best designed interface, users can make mistakes. These mistakes can be both physical (accidentally pointing to the wrong command or data) and mental (making a wrong decision about which command or data to select). An effective design avoids situations that are likely to result in errors. It also accommodates potential user errors and makes it easy for the user to recover.

For information about applying the principle of forgiveness, see User Assistance, which provides information about supporting discoverability in the interface through the use of contextual, task-oriented, and reference forms of user assistance. For information about designing for the widest range of users, see Special Design Considerations.

## Feedback

Always provide feedback for a user's actions. Visual, and sometimes audio, cues should be presented with every user interaction to confirm that the software is responding to the user's input and to communicate details that distinguish the nature of the action.

Effective feedback is timely, and is presented as close to the point of the user's interaction as possible. Even when the computer is processing a particular task, provide the user with information regarding the state of the process and how to cancel that process if that is an option. Nothing is more disconcerting than a "dead" screen that is unresponsive to input. A typical user will tolerate only a few seconds of an unresponsive interface.

It is equally important that the type of feedback you use be appropriate to the task. Pointer changes or a status bar message can communicate simple information; more complex feedback may require the display of a message box.

For information about applying the principle of visual and audio feedback, see Visual Design, and Special Design Considerations.

## Aesthetics

The visual design is an important part of a software's interface. Visual attributes provide valuable impressions and communicate important cues to the interaction behavior of particular objects. At the same time, it is important to remember that every visual element that appears on the screen potentially competes for the user's attention. Provide a pleasant environment that clearly contributes to the user's understanding of the information presented. A graphics or visual designer may be invaluable with this aspect of the design.

For information and guidelines related to the aesthetics of your interface, see Visual Design. This topic covers everything from individual element design to font use and window layout.

## Simplicity

An interface should be simple (not simplistic), easy to learn, and easy to use. It must also provide access to all functionality provided by an application. Maximizing functionality and maintaining simplicity work against each other in the interface. An effective design balances these objectives.

One way to support simplicity is to reduce the presentation of information to the minimum required to communicate adequately. For example, avoid wordy descriptions for command names or messages. Irrelevant or verbose phrases clutter your design, making it difficult for users to easily extract essential information. Another way to design a simple but useful interface is to use natural mappings and semantics. The arrangement and presentation of elements affects their meaning and association.

You can also help users manage complexity by using progressive disclosure. *Progressive disclosure* involves careful organization of information so that it is shown only at the appropriate time. By “hiding” information presented to the user, you reduce the amount of information to process. For example, clicking a menu displays its choices; the use of dialog boxes can reduce the number of menu options.

Progressive disclosure does not imply using unconventional techniques for revealing information, such as requiring a modifier key as the only way to access basic functions or forcing the user down a longer sequence of hierarchical interaction. This can make an interface more complex and cumbersome.

For information about applying the principle of simplicity, see Menus, Controls, and Toolbars. This topic discusses progressive disclosure in detail and describes how and when to use the standard (system-supplied) elements in your interface.

## Design Methodology

Effective interface design is more than just following a set of rules. It requires a user-centered attitude and design methodology. It also involves early planning of the interface and continued work through the software development process.

## A Balanced Design Team

An important consideration in the design of a product is the composition of the team that designs and builds it. Always try to balance disciplines and skills, including development, visual design, writing, human factors, and usability assessment. Rarely are these characteristics found in a single individual, so create a team of individuals who specialize in these areas and who can contribute uniquely to the final design.

Ensure that the design team can effectively work and communicate together. Locating them in close proximity or providing them with a common area to work out design details often fosters better communication and interaction.

## The Design Cycle

An effective user-centered design process involves a number of important phases: designing, prototyping, testing, and iterating. The following sections describe these phases.

## Design

The initial work on a software's design can be the most critical because, during this phase, you decide the general shape of your product. If the foundation work is flawed, it is difficult to correct afterwards.

This part of the process involves not only defining the objectives and features for your product, but understanding who your users are and their tasks, intentions, and goals. This includes understanding factors such as their background — age, gender, expertise, experience level, physical limitations, and special needs; their work environment — equipment, social and cultural influences, and physical surroundings; and their current task organization — the steps required, the dependencies, redundant activities, and the output objective. An order-entry system may have very different users and requirements than an information kiosk.

At this point, begin defining your conceptual framework to represent your product with the knowledge and experience of your target audience. Ideally, you want to create a design model that fits the user's conceptual view of the tasks to be performed. Consider the basic organization and different types of metaphors that can be employed. Observing users at their current tasks can provide ideas on effective metaphors to use.

Document your design. Committing your planned design to a written format not only provides a valuable reference point and form of communication, but often helps make the design more concrete and reveals issues and gaps.

## Prototype

After you have defined a design model, prototype some of the basic aspects of the design. This can be done with “pencil and paper” models — where you create illustrations of your interface to which other elements can be attached; storyboards — comic book-like sequences of sketches that illustrate specific processes; animation — movie-like simulations; or operational software using a prototyping tool or normal development tools.

A prototype is a valuable asset in many ways. First, it provides an effective tool for communicating the design. Second, it can help you define task flow and better visualize the design. Finally, it provides a low-cost vehicle for getting user input on a design. This is particularly useful early in the design process.

The type of prototype you build depends on your goal. Functionality, task flow, interface, operation, and documentation are just some of the different aspects of a product that need to be assessed. For example, pen and paper models or storyboards may work when defining task organization or conceptual ideas. Operational prototypes are usually best for the mechanics of user interaction.

Consider whether to focus your prototype on breadth or depth. The broader the prototype, the more features you should try to include to gain an understanding about how users react to concepts and organization. When your objective is focused more on detailed usage of a particular feature or area of the design, use depth-oriented prototypes that include more detail for a given feature or task.

## Test

User-centered design involves the user in the design process. Usability testing a design, or a particular aspect of a design, provides valuable information and is a key part of a product's success. Usability testing is different than quality assurance testing in that, rather than find programming defects, you assess how well the interface fits user needs and expectations. Of course, defects can sometimes affect how well the interface will fit.

There can be different reasons for testing. You can use testing to look for potential problems in a proposed design. You can also focus on comparative studies of two or more designs to determine which is better, given a specific task or set of tasks.

Usability testing provides you not only with task efficiency and success-or-failure data, it also can provide you with information about the user's perceptions, satisfaction, questions, and problems, which may be just as significant as the ability to complete a particular task.

When testing, it is important to use participants who fit the profile of your target audience. Using fellow workers from down the hall might be a quick way to find participants, but software developers rarely have the same experience as their customers. The section, "Usability Assessment in the Design Process," provides details about conducting a usability test.

## Iterate

Because testing often uncovers design weaknesses, or at least provides additional information you will want to use, repeat the entire process, taking what you have learned and reworking your design or moving onto reprototyping and retesting. Continue this refining cycle through the development process until you are satisfied with the results.

During this iterative process, you can begin substituting the actual application for prototypes as the application code becomes available. However, avoid delaying your design cycle waiting for the application code to be complete enough; you can lose valuable time and input that you could have captured with a prototype. Moreover, by the time most applications are complete enough for testing, it is difficult to consider significant changes, because it becomes easier to ignore usability defects because of the time resources invested. In addition, changes at this point may affect the application's delivery schedule.

## Usability Assessment in the Design Process

As described in the previous section, usability testing is a key part of the design process, but testing design prototypes is only one part of the picture. Usability assessment should begin in the early stages of product development, where you can use it to gather data about how users do their work. You then roll your findings back into the design process. As the design progresses, usability assessment continues to provide valuable input for analyzing initial design concepts and, in the later stages of product development, can be used to test specific product tasks. Apply usability assessment early and often.

Consider the user's entire experience as part of a product's usability. The usability assessment should include all of a product's components. A software interface is more than just what shows up on the screen or in the documentation.

# Usability Testing Techniques

Usability testing involves a wide range of techniques and investment of resources, including trained specialists working in sound-proofed labs with one-way mirrors and sophisticated recording equipment. However, even the simplest investment of an office or conference room, tape recorder, stopwatch, and notepad can produce benefits. Similarly, all tests need not involve great numbers of subjects. More typically, quick, iterative tests with a small, well-targeted sample, 6-10 participants, can identify 80 to 90 percent of most design problems.

Like the design process itself, usability testing begins with defining the target audience and test goals. When designing a test, focus on tasks — not features. Even if your goal is testing specific features, remember that your customers will use them within the context of particular tasks. It is also a good idea to run a pilot test to work out the bugs of the tasks to be tested and make certain the task scenarios, prototype, and equipment work smoothly.

When conducting the usability test, provide an environment comparable to the target setting; usually a quiet location, free from distractions, is best. Make participants feel comfortable. Unless you have participated yourself, you may be surprised by the pressure many test participants feel. You can alleviate some pressure by explaining the testing process and equipment to the participants, and stating your objective in testing the software and not them; if they become confused or frustrated, it is not a reflection upon them.

Allow the user reasonable time to try and work through any difficult situations. Although it is generally best to not interrupt participants during a test, they may get stuck or end up in situations that require intervention. This need not necessarily disqualify the test data, as long as the test coordinator carefully guides or hints around a problem. Give general hints before moving to specific advice. For more difficult situations, you may need to stop the test and make adjustments. Keep in mind that less intervention usually yields better results. Always record the techniques and search patterns that users employ when attempting to work through a difficulty, and the number and type of hints you have to provide.

Ask subjects to think aloud as they work, so you can hear what assumptions and inferences they are making. As the participants work, record the time they take to perform a task as well as any problems they encounter. You may also want to follow up the session with a questionnaire that asks the participants to evaluate the product or tasks they performed.

Record the test results using a portable tape recorder, or better, a video camera. Since even the best observer can miss details, reviewing the data later will prove invaluable. Recorded data also allows more direct comparisons between multiple participants. It is usually risky to base conclusions on observing a single subject. Recorded data also allows all the design team to review and evaluate the results.

Whenever possible, involve *all* members of the design team in observing the test and reviewing the results. This ensures a common reference point and better design solutions as team members apply their own insights to what they observe. If direct observation is not possible, make the recorded results available to the entire team.

## Other Assessment Techniques

There are many techniques you can use to gather usability information. In addition to those already mentioned, “focus groups” are helpful for generating initial ideas or trying out ideas. A focus group requires a moderator who directs the discussion about aspects of a task or design, but allows participants to freely express their opinions. You can also conduct demonstrations, or “walkthroughs,” in which you take the user through a set of sample scenarios and ask about their impressions along the way. In a so-called “Wizard of Oz” technique, a testing specialist simulates the interaction of an interface. Although these latter techniques can be valuable, they often require a trained, experienced test coordinator.

## Understanding Users

The design and usability techniques described in the previous sections have been used in the development of Windows and in many of the guidelines included in this book. That process has yielded the following general characteristics about users. Consider these characteristics in the design of your software:

- Beginning Windows users often have difficulty with the mouse. For example, dragging and double-clicking are skills that may take time for beginning mouse users to master. Dragging can be difficult because it requires continued pressure on the mouse button and involves properly targeting the correct destination. Double-clicking is not the same as two separate clicks, so many beginning users have difficulty handling the timing necessary to distinguish these two actions, or they overgeneralize the behavior to assume that everything needs double-clicking. Design your interface so that double-clicking and dragging are not the only ways to perform basic tasks; allow the user to conduct those tasks using single click operations.
- Beginning users often have difficulty with window management. They do not always realize that overlapping windows represent a three-dimensional space. As a result, when a window is hidden by another, a user may assume it no longer exists.
- Beginning users often have difficulty with file management. The organization of files and folders nested more than two levels is more difficult to understand because it is not as obvious in the real world.
- Intermediate users may understand file hierarchies, but have difficulty with other aspects of file management — such as moving and copying files. This may be because most of their experience working with files is often from within an application.
- Advanced, or “power,” users want efficiency. The challenge in designing for advanced users is providing for efficiency without introducing complexity for less-experienced users. (Shortcut methods are often useful for supporting these users.) In addition, advanced users may be dependent upon particular interfaces, making it difficult for them to adapt to significant rearrangement or changes in an interface.
- To develop for the widest audience, consider international users and users with disabilities. Including these users as part of your planning and design cycle is the best way to ensure that you can accommodate them.

## Design Tradeoffs

A number of additional factors may affect the design of a product. For example, marketing considerations may require you to deliver a product with a minimal design process, or comparative evaluations may force you to consider additional features. Remember that shortcuts and additional features can affect the product. There is no simple equation to determine when a design tradeoff is appropriate. So in evaluating the impact, consider the following:

- Every additional feature potentially affects performance, complexity, stability, maintenance, and the support costs of an application.
- It is harder to fix a design problem after the release of a product because users may adapt, or even become dependent on, a peculiarity in the design.
- Simplicity is not the same as being simplistic. Making something simple to use often requires a good deal of work and code.
- Features implemented by a small extension in the application code do not necessarily have a proportional effect in a user interface. For example, if the primary task is selecting a single object, extending it to support selection of multiple objects could make the frequent, simple task more difficult to carry out.

Input Basics

## Introduction

A user can interact with objects in the interface using different types of input devices. The most common input devices are the mouse, the keyboard, and the pen. This topic covers the basic behavior for these devices; it does not exclude other forms of input.

 Mouse Input

-  Mouse Pointers
-  Mouse Actions

 Keyboard Input

-  Text Keys
  -  Access Keys
-  Mode Keys
-  Shortcut Keys

 Pen Input

-  Pen Pointers
-  Pen Gestures
-  Pen Recognition
-  Ink Input
-  Targeting

# Mouse Input

The mouse is a primary input device for interacting with objects in the Microsoft Windows interface. Other types of pointing devices that emulate a mouse, such as trackballs, fall under the general use of the term “mouse.”

For more information about interactive techniques such as navigation, selection, viewing, editing, transfer, and creating new objects, see General Interaction Techniques.

## Mouse Pointers

The mouse is operationally linked with a graphic on the screen called the *pointer* (also referred to as the *cursor*). By positioning the pointer and clicking the buttons on the mouse, a user can select objects and their operations.

As a user moves the pointer across the screen, its appearance can change to provide feedback about a particular location, operation, or state. Table 4.1 lists some common pointer shapes and their uses.

**Table 4.1 Common Pointers**

| Shape | Screen location | Available or current action |
|---|---|---|
|  | Over most objects | Pointing, selecting, or moving. |
|  | Over text | Selecting text. |
|  | Over any object or location | Processing an operation. |
|  | Over any screen location | Processing in the background (application loading), but the pointer is still interactive. |
|  | Over most objects | Context-sensitive Help mode. |
|  | Inside a window | Zooming a view. |
|  | Over a sizable edge | Resizing an edge vertically. |
|  | Over a sizable edge | Resizing an edge horizontally. |
|  | Over a sizable edge | Resizing an edge diagonally. |
|  | Over a sizable edge | Resizing an edge diagonally. |
|  | Along column gridlines | Resizing a column. |
|  | Along row gridlines | Resizing a row. |
|  | Over split box in | Splitting a window (or vertical scroll bar adjusting a split) horizontally. |
|  | Over split box in | Splitting a window (or horizontal scroll bar adjusting a split) vertically. |
|  | Over any object | Not available as a drop target. |

The system does not provide all of these pointers. For more information about designing your own pointers, see Visual Design.

Each pointer has a particular point — called a *hot spot* — that defines the exact screen location of the mouse. The hot spot determines what object is affected by mouse actions. Screen objects can additionally define a hot zone; the *hot zone* defines the area the hot spot must be within to be considered over the object. Typically, the hot zone coincides with the borders of an object, but it may be larger, or smaller, to make user interaction easier.

## Mouse Actions

Basic mouse actions in the interface use mouse button 1 or button 2. By default, button 1 is the leftmost mouse button and button 2 is the rightmost button. The system allows the user to swap the mapping of the buttons. Button 2 actions typically duplicate functions already accessible with button 1, but provide those functions more efficiently.

For a mouse with three buttons, button 2 is the *rightmost* button, not the center button.

The following are the common behaviors performed with the mouse.

| Action | Description |
|---|---|
| Pointing | Positioning the pointer so it “points to” a particular object on the screen without using the mouse button. Pointing is usually part of preparing for some other interaction. Pointing is often an opportunity to provide visual cues or other feedback to a user. |
| Clicking | Positioning the pointer over an object and then pressing and releasing the mouse button. Generally, the mouse is not moved during the click, and the mouse button is quickly released after it is pressed. Clicking identifies (selects) or activates objects. |
| Double-clicking | Positioning the pointer over an object and pressing and releasing the mouse button twice in rapid succession. Double-clicking an object typically invokes its default operation. |
| Pressing | Positioning the pointer over an object and then holding down the mouse button. Pressing is often the beginning of a click or drag operation. |
| Dragging | Positioning the pointer over an object, pressing down the mouse button while holding the mouse button down, and moving the mouse. Use dragging for actions such as selection and direct manipulation of an object. |

For most mouse interactions, pressing the mouse button only identifies an operation. User feedback is usually provided at this point. Releasing the mouse button activates (carries out) the operation. An auto-repeat function — for example, pressing a scroll arrow to continuously scroll — is an exception.

This guide does not cover other mouse behaviors such as *chording* (pressing multiple mouse buttons simultaneously) and multiple-clicking (triple- or quadruple-clicking). Because these behaviors require more user skill, they are not generally recommended for basic operations. However, you can consider them for special shortcut operations.

Because not every mouse has a third button, there is no basic action defined for a third (middle) mouse button. It is best to limit the assignment of operations to this button to those environments where the availability of a third mouse button can be assumed, and for providing redundant or shortcut access to operations supported elsewhere in the interface. When assigning actions to the button, you need to define the behaviors for the actions already described (pointing, clicking, dragging, and double-clicking) for this button.

# Keyboard Input

The keyboard is a primary means of entering or editing text information. However, the Windows interface also supports the use of keyboard input to navigate, toggle modes, modify input, and, as a shortcut, to invoke certain operations.

For more information about using the keyboard for navigation, selection, and editing, see General Interaction Techniques.

Following are the common interactive behaviors performed with the keyboard.

| Action | Description |
|---|---|
| Pressing | Pressing and releasing a key. Unlike mouse interaction, keyboard interaction occurs upon the down transition of the key. Pressing typically describes the keyboard interaction for invoking particular commands or for navigation. |
| Holding | Pressing and holding down a key. Holding typically describes interaction with keys such as ALT, SHIFT, and CTRL that modify the standard behavior of other input — for example, another key press or mouse action. |
| Typing | Typing input of text information from the keyboard. |

## Text Keys

Text keys include the following:

- Alphanumeric keys (a–z, A–Z, 0–9)
- Punctuation and symbol keys
- TAB and ENTER keys
- The SPACEBAR

In text-entry contexts, pressing a text key enters the corresponding character and typically displays that character on the screen. Except in special views, the characters produced by the TAB and ENTER keys are not usually visible. In some contexts, text keys can also be used for navigation or for invoking specific operations.

Most keyboards include two keys labeled ENTER: one on the main keyboard and one on the numeric keypad. Because these keys have the same label (and on some keyboards the latter may not be available), assign both keys the same functionality.

## Access Keys

An access key is an alphanumeric key — sometimes referred to as a *mnemonic* — that when used in combination with the ALT key navigates to and activates a control. The access key matches one of the characters in the text label of the control. For example, pressing ALT+O activates a control whose label is “Open” and whose assigned access key is “O”. Typically, access keys are not case sensitive. The effect of activating a control depends on the type of control.

Assign access key characters to controls using the following guidelines (in order of choice):

1. The first letter of the label for the control, unless another letter provides a better mnemonic association.
2. A distinctive consonant in the label.
3. A vowel in the label.

Avoid assigning a character where the visual indication of the access key cannot be distinguished from the character. Also, avoid using a character normally assigned to a common function. For example, when you include an Apply button, reserve the “A” — or its localized equivalent — as the access key for that button. In addition, do not assign access keys to the OK and Cancel commands when they map to the ENTER and ESC keys, respectively.

Nonunique access key assignments within the same scope access the first control. Depending on the control, if the user presses the access key a second time, it may or may not access another control with the same assignment. Therefore, define an access key to be unique within the scope of its interaction — that is, the area in which the control exists and to which keyboard input is currently being directed.

Controls without explicit labels can use static text controls to create labels with assigned access keys. Software that supports a nonroman writing system (such as Kanji), but that runs on a standard keyboard, can prefix each control label with an alphabetic (roman) character as its access key.

For more information about static text controls, see Menus, Controls, and Toolbars.

## Mode Keys

Mode keys change the actions of other keys (or other input devices). There are two kinds of mode keys: toggle keys and modifier keys.

A toggle key turns a particular mode on or off each time it is pressed. For example, pressing the CAPS LOCK key toggles uppercase alphabetic keys; pressing the NUM LOCK key toggles between numeric and directional input using the keypad keys.

Like toggle keys, modifier keys change the actions of normal input. Unlike toggle keys, however, modifier keys establish modes that remain in effect only while the modifier key is held down. Modifier keys include the SHIFT, CTRL, and ALT keys. Such a “spring-loaded” mode is often preferable to a “locked” mode because it requires the user to continuously activate it, making it a conscious choice and allowing the user to easily cancel the mode by releasing the key.

Because it can be difficult for a user to remember multiple modifier assignments, avoid using multiple modifier keys as the primary means of access to basic operations. In some contexts, such as environments that are specific to pen input, the keyboard may not be available. Therefore, use modifier-based actions only for quick access to operations that are supported adequately elsewhere in the interface.

# Shortcut Keys

Shortcut keys (also referred to as accelerator keys) are keys or key combinations that, when pressed, provide quick access to frequently performed operations. CTRL+*letter* combinations and function keys (F1 through F12) are usually the best choices for shortcut keys. By definition, a shortcut key is a keyboard equivalent of functionality that is supported adequately elsewhere in the interface. Therefore, avoid using a shortcut key as the only way to access a particular operation.

Function key and modified function key combinations may be easier for international users because they have no mnemonic relationship. However, there is a tradeoff because function keys are often more difficult to remember and to reach. For a list of the most common shortcut key assignments, see Keyboard Interface Summary.

When defining shortcut keys, observe the following guidelines:

- Assign single keys where possible because these keys are the easiest for the user to perform.
- Make modified-letter key combinations case insensitive.
- Use SHIFT+*key* combinations for actions that extend or complement the actions of the key or key combination used without the SHIFT key. For example, ALT+TAB switches windows in a top-to-bottom order. SHIFT+ALT+TAB switches windows in reverse order. However, avoid SHIFT+*text* keys, because the effect of the SHIFT key may differ for some international keyboards.
- Use CTRL+*key* combinations for actions that represent a larger scale effect. For example, in text editing contexts, HOME moves to the beginning of a line, and CTRL+HOME moves to the beginning of the text. Use CTRL+*key* combinations for access to commands where a letter key is used — for example, CTRL+B for bold. Remember that such assignments may be meaningful only for English-speaking users.
- Avoid ALT+*key* combinations because they may conflict with the standard keyboard access for menus and controls. The ALT+*key* combinations — ALT+TAB, ALT+ESC, and ALT+SPACEBAR — are reserved for system use. ALT+*number* combinations enter special characters.
- Avoid assigning shortcut keys defined in this guide to other operations in your software. That is, if CTRL+C is the shortcut for the Copy command and your application supports the standard copy operation, don't assign CTRL+C to another operation.
- Provide support for allowing the user to change the shortcut key assignments in your software, when possible.
- Use the ESC key to stop a function in process or to cancel a direct manipulation operation. It is also usually interpreted as the shortcut key for a Cancel button.

Some keyboards also support three new keys, the Application key and the two Windows keys. The primary use for the Application key is to display the pop-up menu for current selection (same as SHIFT+F10). You may also use it with modifier keys for application-specific functions. Pressing either of the Windows keys — left or right —displays the Start menu. These keys are also used by the system as modifiers for system-specific functions. Do not use these keys as modifiers for nonsystem-level functions.

# Pen Input

Systems with a Windows pen driver installed support user input using tapping or writing on the surface of the screen or a tablet with a pen, and in some cases with a finger.

The **GetSystemMetrics** function provides access to the SM_PENWINDOWS constant that indicates when a pen is installed. For more information about this function, see the documentation included in the Microsoft Win32 Software Development Kit (SDK).

Depending on the placement of the pen, you can use it for both pointing and writing. For example, if you move the pen over menus or most controls, it acts as a pointing device. Because of the pointing capabilities of the pen, the user can perform most mouse-based operations. When over a text entry or drawing area, the pen becomes a writing or drawing tool; the pointer changes to a pen shape to provide feedback to the user. When the tip of the pen touches the input surface, the pen starts *inking* — that is, tracing lines on the screen. The user can then draw shapes, characters, and other patterns; these patterns remain on the screen exactly as drawn or can be recognized, interpreted, and redisplayed.

The pen can retain the functionality of a pointing device (such as a mouse) even in contexts where it would normally function as a writing or drawing tool. For example, you can use timing to differentiate operations; that is, if the user holds the pen tip in the same location for a predetermined period of time, a different action may be inferred. However, this method is often unreliable or inefficient for many operations, so it may be better to use toolbar buttons to switch to different modes of operation. Choosing a particular button allows the user to define whether to use the pen for entering information (writing or drawing) or as a pointing device.

You can also provide the user with access to other operations using an action handle. An *action handle* is a special graphic displayed for a selection. An action handle can be used to support direct manipulation operations or to provide access to pop-up menus.

For more information about action handles, see General Interaction Techniques.

Following are the fundamental behaviors defined for a pen.

| Action | Description |
|---|---|
| Pressing | Positioning and pressing the tip to the input surface. A pen press is equivalent to a mouse press and typically identifies a particular pen action. |
| Tapping | Pressing the pen tip on the input surface and lifting it without moving the pen. In general, tapping is equivalent to clicking mouse button 1. Therefore, this action typically selects an object, setting a text insertion point or activating a button |
| Double-tapping | Pressing and lifting the pen tip twice in rapid succession. Double-tapping is usually interpreted as the equivalent to double-clicking mouse button 1. |
| Dragging | Pressing the pen tip on the input surface and keeping it pressed while moving the pen. In inking contexts, you can use dragging for the input of pen strokes for writing, drawing, gestures, or for direct manipulation, depending on which is most appropriate for the context. In noninking contexts, it is the equivalent of a mouse drag. |

A user may move the pen more between taps when double-tapping than a user double-clicking with a mouse. As a result, you may want to slightly increase your hot zones for detecting a double-tap when

of a pen device has been installed.

Some pens include buttons on the pen barrel that can be pressed. For pens that support barrel buttons, the following behaviors may be supported.

| Action | Description |
| --- | --- |
| Barrel-tapping | Holding down the barrel button of the pen while tapping. Barrel-tapping is equivalent to clicking with mouse button 2. |
| Barrel-dragging | Holding down the barrel button of the pen while dragging the pen. Barrel-dragging is equivalent to dragging with mouse button 2. |

Because not all pens support barrel buttons, any behaviors that you support using a barrel button should also be supported by other techniques in the interface.

Pen input is delimited, by the lifting of the pen tip, an explicit termination tap (such as tapping the pen on another window or as the completion of a gesture), or a time-out without further input. You can also explicitly define an application-specific recognition time-out.

*Proximity* is the ability to detect the position of the pen without it touching the input surface. While Windows provides support for pen proximity, avoid depending on proximity as the exclusive means of access to basic functions, because not all pen hardware supports this feature. Even pen hardware that does support proximity may allow other non-pen input, such as touch input, where proximity cannot be supported.

## Pen Pointers

For pen tablets, as with a mouse, pointers play an important part in visually indicating the user's location of interaction on the screen. When the input surface is actually a screen display, pointers may seem superfluous; however, they still have an important role to play. Pointers help the pen user select small targets faster. Moreover, changes from one pointer to another provide useful feedback about the actions supported by the object under the pen. For example, when the pen moves over a resizable border, the pointer can change from a pen (indicating that writing is possible) to a resizing pointer (indicating that the border can be dragged to resize the object). Whenever possible, include this type of feedback in pen-enabled applications to help users understand the kinds of supported actions.

Following are two common pointers used with the pen.

**Table 4.2 Pen Pointers**

| Shape | Common usage |
|---|---|
|  | Pointing, selecting, moving, and resizing |
|  | Writing and drawing |

When the screen is the input surface — because a pointer may be partially obscured by the pen or by the user's hand — you may need to consider including additional forms of feedback, such as toolbar button states or status bar information, to indicate the pen's input state.

# Pen Gestures

When using the pen for writing, certain ink patterns are interpreted as *gestures*. Using one of these specially drawn symbols invokes a particular operation, such as deleting text, or produces a nonprinting text character, such as a carriage return or a tab. For example, a circled X gesture is equivalent to the Cut command. After the system interprets a gesture, the gesture's ink is removed from the display.

For more information about common gestures and their interpretation, see General Interaction Techniques.

All gestures include a circular stroke to distinguish them from ordinary characters. Most gestures also operate positionally; in other words, they act upon the objects on which they are drawn. Determining the position of the specific gesture depends on either the area surrounded by the gesture or a single point — the hot spot of the gesture.

Pen gestures usually cannot be combined with ink (writing or drawing actions) within the same recognition sequence. For example, the user cannot draw a few characters, immediately followed by a gesture, followed by more characters.

The rapidity of gestural commands is one of the key advantages of the pen. Do not rely on gestures as the only or primary way to perform commands, however, because gestures require memorization by users. Regard gestures as a quick access, shortcut method for operations adequately supported elsewhere in the interface, such as in menus or buttons. If the pen extensions are installed, you can optionally place a bitmap of the gesture next to the corresponding command (in place of the keyboard shortcut text) to help the user learn particular gestures.

In addition, avoid using gestures when they interfere with common functionality or make operations with parallel input devices, such as the mouse or keyboard, more cumbersome. For example, although writing a character gesture in a list box could be used as a way to scroll automatically within the list, it would interfere with the basic and more frequent user action of selecting an item in the list. A better technique is to provide a text input field where the user can write and, based on the letters entered, scroll the list.

# Pen Recognition

*Recognition* is the interpretation of pen strokes into some standardized meaning. Consider recognition as a means to an end, not an end in itself. Do not use recognition if it is unnecessary or if it is not the best interface. For example, it may be more effective to provide a control that allows a user to select a date, rather than requiring the user to write it in just so your software can recognize it.

Accurate recognition is difficult to achieve, but you can greatly improve your recognition interface by providing a fast, easy means to correct errors. For example, if you allow users to overwrite characters or choose alternatives, they will be less frustrated and find recognition more useful. You can also improve recognition by using context and constraints. For example, a checkbook application can constrain certain fields to contain only numbers.

## Ink Input

In some cases — for example, signatures — recognition of pen input may be unnecessary; the ink is a sufficient representation of information. Ink is a standard data type supported by the Clipboard. Consider supporting ink entries as input wherever your software accepts normal text input, unless the representation of that input needs to be interpreted for other operations, such as searching or sorting.

## Targeting

Targeting, or determining where to direct pen input, is an important design factor for pen-enabled software. For example, if the user gestures over a set of objects, which objects should be affected? If the user writes text that spans several writing areas, which text should be placed in which area? In general, you use the context of the input to determine where to apply pen input. More specifically, use the following guidelines for targeting gestures on objects:

- If the user draws the gesture on any part of a selection, apply the gesture to the selection.
- If the user draws the gesture on an object that is not selected, select that object, and apply the gesture to that object.
- If the user does not draw the gesture on any object or selection, but there is a selection, apply the gesture to that selection.

If none of these guidelines applies, ignore the gesture.

For handwriting, you can also use context to determine where to direct the input. Figure 4.1 demonstrates how the proximity of the text to the text boxes determines the destination of the written text.

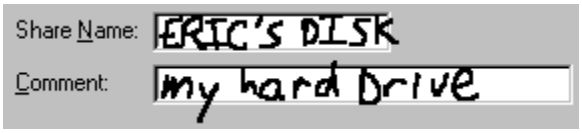



**Figure 4.1 Targeting handwritten input**

The system's pen services provide basic support for targeting, but your application can also provide additional support. For example, your application can define a larger inking rectangle than the control usually provides. In addition, because your application often knows the type of input to expect, it can use this information to better interpret where to target the input.

# Introduction

This topic covers basic interaction techniques, such as navigation, selection, viewing, editing, and creation. Many of these techniques are based on an object-action paradigm in which a user identifies an object and an action to apply to that object. By maintaining these techniques consistently, you enable users to transfer their skills to new tasks.

Where applicable, support the basic interaction techniques for the mouse, keyboard, and pen. When adding or extending these basic techniques, consider how the feature or function can be supported across input devices. Techniques for a particular device need not be identical for all devices. Instead, tailor techniques to optimize the strengths of a particular device. In addition, make it easy for the user to switch between devices so that an interaction started with one device can be completed with another.

## Navigation

One of the most common ways of identifying or accessing an object is by navigating to it. The following sections include information about mouse, pen, and keyboard techniques.

## Mouse and Pen Navigation

Navigation with the mouse is simple; when a user moves the mouse left or right, the pointer moves in the corresponding direction on the screen. As the mouse moves away from or toward the user, the pointer moves up or down. By moving the mouse, the user can move the pointer to any location on the screen. Pen navigation is similar to mouse navigation, except that the user navigates by moving the pen without touching the input surface.

## Keyboard Navigation

Keyboard navigation requires a user to press specific keys and key combinations to move the *input focus* — the indication of where the input is being directed — to a particular location. The appearance of the input focus varies by context; in text, it appears as a text cursor or insertion point.

For more information about displaying the input focus, see Visual Design.

## Basic Navigation Keys

The navigation keys are the four arrow keys and the HOME, END, PAGE UP, PAGE DOWN, and TAB keys. Pressed in combination with the CTRL key, a navigation key increases the movement increment. For example, where pressing RIGHT ARROW moves right one *character* in a text field, pressing CTRL+RIGHT ARROW moves right one *word* in the text field. Table 5.1 lists the common navigation keys and their functions. You can define additional keys for navigation.

**Table 5.1 Basic Navigation Keys**

| Key | Moves cursor to | CTRL+key moves cursor to |
|---|---|---|
| LEFT ARROW | Left one unit. | Left one (larger) unit. |
| RIGHT ARROW | Right one unit. | Right one (larger) unit. |
| UP ARROW | Up one unit or line. | Up one (larger) unit. |
| DOWN ARROW | Down one unit or line. | Down one (larger) unit. |
| HOME | Beginning of line. | Beginning of data or file (topmost position). |
| END | End of line. | End of data or file (bottommost position). |
| PAGE UP | Up one screen (previous screen, same position). | Left one screen (or previous unit, if left is not meaningful). |
| PAGE DOWN | Down one screen (next screen, same position). | Right one screen (or next unit, if right is not meaningful). |
| TAB | Next field. (SHIFT+TAB moves in reverse order). | Next larger field. |

For more information about keyboard navigation in secondary windows, such as dialog boxes, see Secondary Windows.

Unlike mouse and pen navigation, keyboard navigation typically affects existing selections. Optionally, you can support the SCROLL LOCK key to enable scrolling navigation without affecting existing selections. If you do so, the keys scroll the appropriate increment.

## Selection

Selection is the primary means by which the user identifies objects in the interface. Consequently, the basic model for selection is one of the most important aspects of the interface.

Selection typically involves an overt action by the user to identify an object. This is known as an *explicit selection*. Once the object is selected, the user can specify an action for the object.

There are also situations where the identification of an object can be derived by inference or implied by context. An *implicit selection* works most effectively where the association of object and action is simple and visible. For example, when the user drags a scroll box, the user establishes selection of the scroll box and the action of moving at the same time. Implicit selection may result from the relationships of a particular object. For example, selecting a character in a text document may implicitly select the paragraph of which the character is a part.

A selection can consist of a single object or multiple objects. Multiple selections can be *contiguous* — where the selection set is made up of objects that are logically adjacent to each other, also known as a *range selection*. A *disjoint selection* set is made up of objects that are spatially or logically separated.

Multiple selections may also be classified as *homogeneous* or *heterogeneous*, depending on the type or properties of the selected objects. Even a homogeneous selection might have certain aspects in which it is heterogeneous. For example, a text selection that includes bold and italic text can be considered homogeneous with respect to the basic object type (characters), but heterogeneous with respect to the values of its font properties. The homogeneity or heterogeneity of a selection affects the access of the operations or properties of the objects in the selection.

## Selection Feedback

Always provide visual feedback for explicit selections as the user makes the selection, so that the user can tell the effect of the selection operation. Display the appropriate selection appearance for each object included in the selection set. The form of selection appearance depends on the object and its context.

For more information about how to visually render the selection appearance of an object, see Visual Design. For more information about how the context of an object can affect its selection appearance, see Working with OLE Embedded and OLE Linked Objects.

You may not need to provide immediate selection feedback for implicit selection; you can often indicate the effects of implicit selection in other ways. For example, when the user drags a scroll box, the scroll box moves with the pointer. Similarly, if the effect of selecting a word in a paragraph implicitly selects the paragraph, you would not use selection appearance on the entire paragraph, but rather reflect the implicit selection by including the paragraph's properties when the user chooses the Properties command.

## Scope of Selection

The *scope* of a selection is the area, extent, or region in which, if other selections are made, they will be considered part of the same selection set. For example, you can select two document icons in the same folder window. However, the selection of these icons is independent of the selection of the window's scroll bar, a menu, the window itself, or selections made in other windows. So, the selection scope of the icons is the area viewed through that window. Selections in different scopes are independent of each other. For example, selections in one window are typically independent of selections in other windows. Their windows define the scope of each selection independently. The scope of a selection is important because you use it to define the available operations for the selected items and how the operations are applied.

## Hierarchical Selection

Range selections typically include objects at the same level. However, you can also support a user's elevating a range selection to the next higher level if it extends beyond the immediate containment of the object (but within the same window). When the user adjusts the range back within the containment of the start of the range, return the selection to the original level. For example, extending a selection from within a cell in a table to the next cell, as shown in Figure 5.1, should elevate the selection from the character level to the cell level; adjusting the selection back within the cell should reset the selection to the character level.

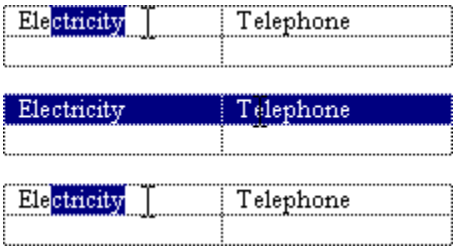



**Figure 5.1 Hierarchical selection**

## Mouse Selection

Selection with the mouse relies on the basic actions of clicking and dragging. In general, clicking selects a single item or location, and dragging selects a single range consisting of all objects logically included from the button-down to the button-up location. If you also support dragging for object movement, use keyboard-modified mouse selection or region selection to support multiple selection.

## Basic Selection

Support user selection using either mouse button. When the user presses the mouse button, establish the starting point, or *anchor point*, of a selection. If, while pressing the mouse button, the user drags the mouse, extend the selection to the object nearest the hot spot of the pointer. If, while continuing to hold the mouse button down, the user drags the mouse within the selection, reduce the selection to the object now nearest the pointer. Tracking the selection with the pointer while the mouse button continues to be held down allows the user to adjust a range selection dynamically. Use appropriate selection feedback to indicate the objects included in the selection.

For more information aboutthe appearance of selection feedback, see Visual Design.

The release of the mouse button ends the selection operation and establishes the *active end* of the selection. If the user presses mouse button 2 to make a selection, display the contextual pop-up menu for the selected objects when the user releases the mouse button.

For more information about pop-up menus, see Menus, Controls, and Toolbars.

The most common form of selection optimizes for the selection of a single object or a single range of objects. In such a case, creating a new selection within the scope of an existing selection (for example, within the same area of the window) cancels the selection of the previously selected objects. This allows simple selections to be created quickly and easily.

When using this technique, reset the selection when the user presses the mouse button and the pointer (hot spot) is outside (not on) any existing selection. If the pointer is over a selected item, however, don't cancel the former selection. Instead, determine the appropriate result according to whether the user pressed mouse button 1 or 2.

If the user presses mouse button 1 and the pointer does not move from the button down point, the effect of the release of the mouse button is determined by the context of the selection. You can support whichever of the following best fits the nature of the user's task:

- The result may have no effect on the existing selection. This is the most common and safest effect.
- The object under the pointer may receive some special designation or distinction; for example, become the next anchor point or create a subselection.
- The selection can be reset to be only the object under the pointer.

If the user pressed mouse button 2, the selection is not affected, but you display a pop-up menu for selection.

Although selection is typically done by positioning the pointer over an object, it may be inferred based on the logical proximity of an object to a pointer. For example, when selecting text, the user can place the pointer on the blank area beyond the end of the line and the resulting selection is inferred as being the end of the line.

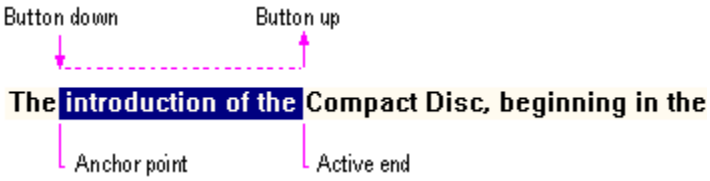
## Selection Adjustment

Selections are adjusted (elements added to or removed from the selection) using keyboard modifiers with the mouse. The CTRL key is the disjoint, or toggle, modifier. If the user presses the CTRL key while making a new selection, preserve any existing selection within that scope and reset the anchor point to the new mouse button-down location. Toggle the selection state of the object under the pointer — that is, if it is not selected, select it; if it is already selected, unselect it.

Disjoint selection techniques may not apply to all situations where you support selection.

If a selection modified by the CTRL key is made by dragging, the selection state is applied for all objects included by the drag operation (from the anchor point to the current pointer location). This means if the first item included during the drag operation is not selected, select all objects included in the range. If the first item included was already selected, unselect it and all the objects included in the range regardless of their original state.

For example, the user can make an initial selection by dragging.




The user can then press the CTRL key and drag to create a disjoint selection, resetting the anchor point.


The user must press the CTRL key before using the mouse button for a disjoint (toggle) selection. After a disjoint selection is initiated, it continues until the user releases the mouse button (even if the user releases the CTRL key before the mouse button).

The SHIFT key adjusts (or extends) a single selection or range selection. When the user presses the mouse button while holding down the SHIFT key, reset the active end of a selection from the anchor point to the location of the pointer. Continue tracking the pointer, resetting the active end as the user drags, similar to a simple range drag selection. When the user releases the mouse button, the selection operation ends. You should then set the active end to the object nearest to the mouse button release point. Do not reset the anchor point. It should remain at its current location.

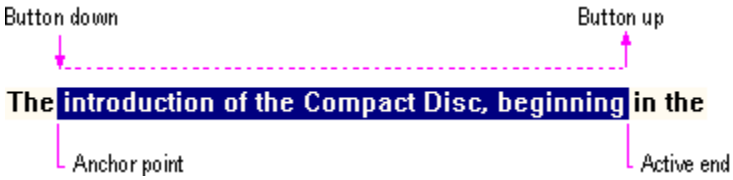
Only the selection made from the current anchor point is adjusted. Any other disjoint selections are not affected unless the extent of the selection overlaps an existing disjoint selection.

The effect on the selection state of a particular object is based on the first item included in the selection range. If the first item is already selected, select (not toggle the selection state of) all objects included in the range; otherwise, unselect (not toggle the selection state of) the objects included.

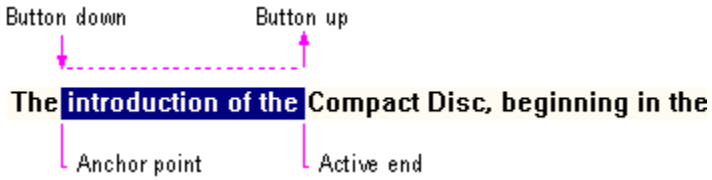
The user must press and hold down the SHIFT key before pressing the mouse button for the action to be interpreted as adjusting the selection. When the user begins adjusting a selection by pressing the SHIFT key, continue to track the pointer and adjust the selection (even if the user releases the modifier key) until the user releases the mouse button.

Pressing the SHIFT modifier key always adjusts the selection from the current anchor point. This means

the user can always adjust the selection range of a single selection or CTRL key–modified disjoint selection. For example, the user can make a range selection by dragging.

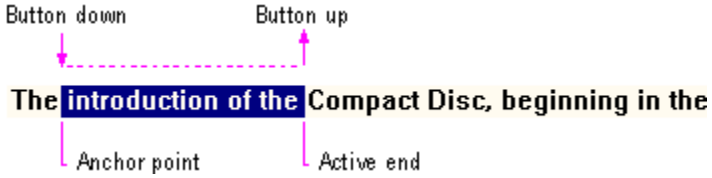



The same result can be accomplished by making an initial selection.




The user can adjust the selection with the SHIFT key and dragging.


The following sequence illustrates how the user can use the SHIFT key and dragging to adjust a disjoint selection. The user makes the initial selection by dragging.




The user presses the CTRL key and drags to create a disjoint selection.


The user can then extend the disjoint selection using the SHIFT key and dragging. This adjusts the selection from the anchor point to the button down point and tracks the pointer to the button up point.

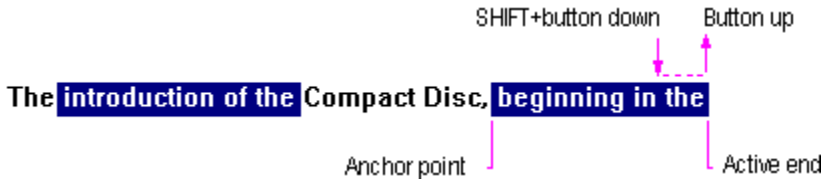



Figure 5.2 shows how these same techniques can be applied within a spreadsheet.

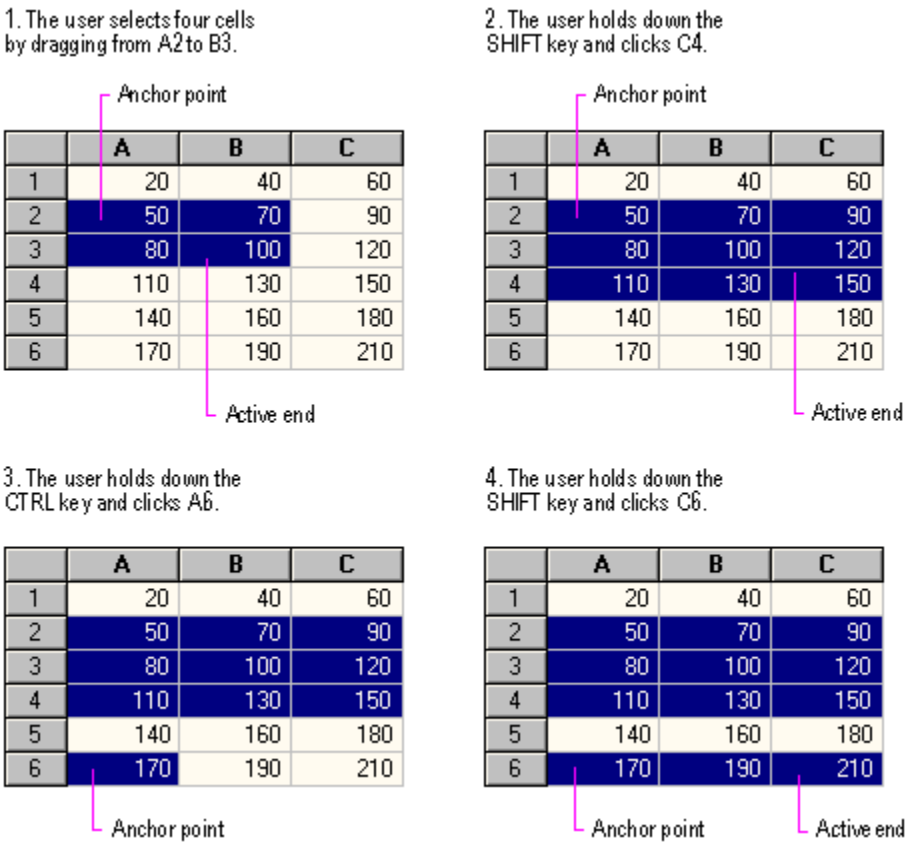



**Figure 5.2 Selection within a spreadsheet**

The following summarizes the mouse selection operations.

| Operation | Mouse action |
|---|---|
| Select object (range of objects) | Click (drag) |
| Disjoint selection state of noncontiguous | CTRL+click (drag)object (range of objects) |
| Adjust current selection to object | SHIFT+click (drag)(or range of objects) |

For more information about the mouse interface, including selection behavior, see Mouse Interface Summary.

## Region Selection

In Z-ordered, or layered, contexts, in which objects may overlap, user selection can begin on the background (sometimes referred to as *white space*). To determine the range of the selection in such cases, a bounding outline (sometimes referred to as a marquee) is drawn. The outline is typically a rectangle, but other shapes (including freeform outline) are possible.

When the user presses the mouse button and moves the pointer (a form of selection by dragging), display the bounding outline, as shown in Figure 5.3. You set the selection state of objects included by the outline using the selection guidelines described in the previous sections, including operations that use the SHIFT and CTRL modifier keys.

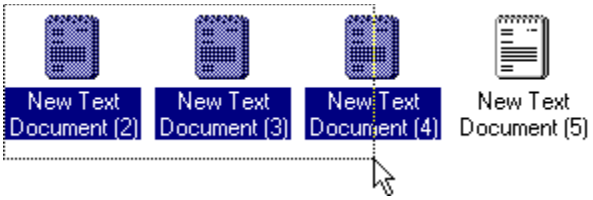



**Figure 5.3 Region selection**

You can use the context of your application and the user's task to determine whether an object must be totally enclosed or only intersected by the bounding region to be affected by the selection operation. Always provide good selection feedback during the operation to communicate to the user which method you support. When the user releases the mouse button, remove the bounding region, but retain the selection feedback.

## Pen Selection

When the pen is being used as the pointing device, you can use the same selection techniques defined for the mouse. For example, in text input controls, you support user selection of text by dragging through it. Standard pen interfaces also support text selection using a special pen selection handle. In discrete object scenarios, like drawing programs, you support selection of individual objects by tapping or by performing region selection by dragging.

For more information about supporting selection in pen-enabled controls, see Pen-Specific Editing Techniques.

In some specialized contexts, you can also use a press-hold-drag technique or the *lasso-tap* gesture to support selection of individual objects or ranges of objects. However, avoid implementing these techniques when it might interfere with primary operations such as direct manipulation. In general, consider using a pen selection handle or pen controls that include the selection handles before you consider these methods.

For the press-hold-drag technique, you switch to a selection mode when the user holds the pen tip at the same location for a predefined time-out. Then the user can drag to make a selection.

Lasso-tap involves making a circular gesture around the object, then tapping within the gesture. For example, in Figure 5.4, making the lasso-tap gesture selects the word “controversial.”

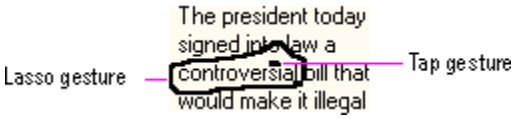



**Figure 5.4 A lasso-tap gesture**

In text contexts, base the selection on the extent of the lasso gesture and the character-word-paragraph granularity of the text elements covered. For example, if the user draws the lasso around a single character, select only that character. If the user draws the lasso around multiple characters within a word, select the entire word. If the gesture encompasses characters in multiple words, select the range of words logically included by the gesture. This reduces the need for the user to be precise.

## Keyboard Selection

Keyboard selection relies on the input focus to define selected objects. The input focus can be an insertion point, a dotted outline box, or some other cursor or visual indication of the location where the user is directing keyboard input.

For more information about input focus, see Visual Design.

In some contexts, selection may be implicit with navigation. When the user presses a navigation key, you move the input focus to the location (as defined by the key) and automatically select the object at that location.

In other contexts, it may be more appropriate to move the input focus and require the user to make an explicit selection with the Select key. The recommended keyboard Select key is the SPACEBAR, unless this assignment directly conflicts with the specific context — in which case, you can use CTRL+SPACEBAR. (If this conflicts with your software, define another key that best fits the context.) In some contexts, pressing the Select key may also unselect objects; in other words, it will toggle the selection state of an object.

## Contiguous Selection

In text contexts, the user moves the insertion point to the desired location using the navigation keys. Set the anchor point at this location. When the user presses the SHIFT key with any navigation key (or navigation key combinations, such as CTRL+END), set that location as the active end of the selection and select all characters between the anchor point and the active end. (Do not move the anchor point.) If the user presses a subsequent navigation key, cancel the selection and move the insertion point to the appropriate location defined by the key. If the user presses LEFT ARROW or RIGHT ARROW keys, move the insertion point to the end of the former selection range. If UP ARROW or DOWN ARROW are used, move the insertion point to the previous or following line at the same relative location.

You can use this technique in other contexts, such as lists, where objects are logically contiguous. However, in such situations, the selection state of the objects logically included from the anchor point to the active end depend on the selection state of the object at, or first traversed from, the anchor point. For example, if the object at the anchor point is selected, then select all the objects in the range regardless of their current state. If the object at the anchor point is not selected, unselect all the items in the range.

## Disjoint Selection

You use the Select key for supporting disjoint selections. The user uses navigation keys or navigation keys modified by the SHIFT key to establish the initial selection. The user can then use navigation keys to move to a new location and subsequently use the Select key to create an additional selection.

In some situations, you may prefer to optimize for selection of a single object or single range. In such cases, when the user presses a navigation key, reset the selection to the location defined by the navigation key. Creating a disjoint selection requires supporting the Add mode key (SHIFT+F8). In this mode, you move the insertion point when the user presses navigation keys without affecting the existing selections or the anchor point. When the user presses the Select key, toggle the selection state at the new location and reset the anchor point to that object. At any point, the user can use the SHIFT+navigation key combination to adjust the selection from the current anchor point.

When the user presses the Add mode key a second time, you toggle out of the mode, preserving the selections the user created in Add mode. But now, if the user makes any new selections within that selection scope, you return to the single selection optimization — canceling any existing selections — and reset the selection to be only the new selection.

## Selection Shortcuts

Double-clicking with mouse button 1 and double-tapping — its pen equivalent — is a shortcut for the default operation of an object. In text contexts, it is commonly assigned as a shortcut to select a word. When supporting this shortcut, select the word and the space following the word, but not the punctuation marks.

Double-clicking as a shortcut for selection only applies to text. In other contexts, it may perform other operations.

You can define additional selection shortcuts or techniques for specialized contexts. For example, selecting a column label may select the entire column. Because shortcuts cannot be generalized across the user interface, however, do not use them as the only way to perform a selection.

## Common Conventions for Supporting Operations

There are many ways to support operations for an object, including direct manipulation of the object or its control point (handle), menu commands, buttons, dialog boxes, tools, or programming. Support for a particular technique is not exclusive to other techniques. For example, the user can size a window by using the Size menu command and by dragging its border.

Design operations or commands to be *contextual*, or related to, the selected object to which they apply. That is, determine which commands or properties, or other aspects of an object, are made accessible by the characteristics of the object and its context (relationships). Often the context of an object may add to or suppress the traits of the object. For example, the menu for an object may include commands defined by the object's type and commands supplied by the object's current container.

## Operations for a Multiple Selection

When determining which operations to display for a multiple selection, use an intersection of the operations that apply to the members of that selection. The selection's context may add to or filter out the available operations or commands displayed to the user.

It is also possible to determine the effect of an operation for a multiple selection based upon a particular member of that selection. For example, when the user selects a set of graphic objects and chooses an alignment command, you can make the operation relative to a particular item identified in the selection.

Limit operations on a multiple selection to the scope of the selected objects. For example, deleting a selected word in one window should not delete selections in other windows (unless the windows are viewing the same selected objects).

## Default Operations and Shortcut Techniques

An object can have a default operation; a *default operation* is an operation that is assumed when the user employs a shortcut technique, such as double-clicking or drag and drop. For example, double-clicking a folder displays a window with the content of the folder. In text editing situations, double-clicking selects the word. The behavior differs because the default commands in each case differ: for a folder, the default command is Open; and for text, it is Select Word.

Similarly, when the user drags and drops an object at a new location with mouse button 1, there must be a default operation defined to determine the result of the operation. Dragging and dropping to some locations can be interpreted as a move, copy, link, or some other operation. In this case, the drop destination determines the default operation.

For more information about supporting default operations for drag and drop, see Transfer Operations; also see Working with OLE Embedded and OLE Linked Objects.

Shortcut techniques for default operations provide greater efficiency in the interface, an important factor for more experienced users. However, because they typically require more skill or experience and because not all objects may have a default operation defined, avoid shortcut techniques as the exclusive means of performing a basic operation. For example, even though double-clicking opens a folder icon, the Open command appears on its menu.

## View Operations

Following are some of the common operations associated with viewing objects. Although these operations may not always be used with all objects, when supported, they should follow similar conventions.

| Operation | Action |
|---|---|
| Open | Opens a primary window for an object. For container objects, such as folders and documents, this window displays the content of the object. |
| Close | Closes a window. |
| Properties | Displays the properties of an object in a window, typically in a property sheet window. |
| Help | Displays a window with the contextual Help information about an object. |

When the user opens a new window, you should display it at the top of the Z order of its peer windows and activate it. Primary windows are typically peers with each other. Display supplemental or secondary windows belonging to a particular application at the top of their local Z order — that is, the Z order of the windows of that application, not the Z order of other primary windows.

For more information about opening windows, property sheets, and Help windows, see Windows, Secondary Windows, and User Assistance, respectively.

If the user interacts with another window before the new window opens, the new window does not appear on top; instead, it appears where it would usually be displayed if the user activated another window. For example, if the user opens window A, then opens window B, window B appears on top of window A. If the user clicks back in window A before window B is displayed, however, window A remains active and at the top of the Z order; window B appears behind window A.

Whether opening a window allows the user to also edit the information in that window's view depends on a number of factors. These factors can include who the user is, the type of view being used, and the content being viewed.

After the user opens a window, re-executing the command that opened the window should activate the existing window instead of opening another instance of the window. For example, if the user chooses the Properties command for a selected object whose property sheet is already open, the existing property sheet is activated, rather than a second window opened.

This guideline applies per user desktop. Two users opening a window for the same object on a network can each see separate windows for the object from their individual desktops.

Closing a window does not necessarily mean quitting the processes associated with the object being viewed. For example, closing a printer's window does not cancel the printing of documents in its queue. Quitting an application closes its windows, but closing a window does not necessarily quit an application. Similarly, you can use other commands in secondary windows which result in closing the window — for example, OK and Cancel. However, the effect of closing the window with a Close command depends on the context of the window. Avoid assuming that the Close command is the equivalent of the Cancel command.

If there are changes transacted in a window that have not yet been applied and the user chooses the Close command, and those changes will be lost if not applied, display a message asking whether the user wishes to apply or discard the changes or cancel the Close operation. If there are no outstanding changes or if pending changes are retained for the next time the window is opened, remove the window.

## View Shortcuts

Following are the recommended shortcut techniques for the common viewing commands.

| Shortcut | Operation |
|---|---|
| CTRL+O | Opens a primary window for an object. For container objects, such as folders and documents, this window displays the content of the object. |
| ALT+F4 | Closes a window. |
| F1 | Displays a window with contextual Help information. |
| SHIFT+F1 | Starts context-sensitive Help mode. |
| Double-click | Carries out the default command.(button 1) or ENTER |
| ALT+double-click | Displays the properties of an object in a or ALT+ENTER window, typically in a property sheet window. |

For more information on reserved and recommended shortcut keys, see Keyboard Interface Summary.

Use double-clicking and the ENTER key to open a view of an object when that view command is the default command for the object. For example, double-clicking a folder opens the folder's primary window. But double-clicking a sound object plays the sound; this is because the Open command is the default command for folders, and the Play command is the default command for sound objects.

## Editing Operations

Editing involves changing (adding, removing, replacing) some fundamental aspect about the composition of an object. Not all changes constitute editing of an object, though. For example, changing the view of a document to an outline or magnified view (which has no effect on the content of the document) is not editing. The following sections cover some of the common interface techniques for editing objects.

## Editing Text

Editing text requires that you target the input focus at the text to be edited. For mouse input, the input focus always coincides with the pointer (button down) location. For the pen, it is the location of the pointer when the pen touches the input surface. For the keyboard, the input focus is determined with the navigation keys. In all cases, the visual indication that a text field has the input focus is the presence of the text cursor, or insertion point.

## Inserting Text

Inserting text involves the user placing the insertion point at the appropriate location and then typing. For each character typed, your application should move the insertion point one character to the right (or left, depending on the language).

If the text field supports multiple lines, *wordwrap* the text; that is, automatically move text to the next line as the textual input exceeds the width of the text-entry area.

## Overtype Mode

Overtype is an optional text-entry behavior that operates similarly to the insertion style of text entry, except that you replace existing characters as new text is entered — with one character being replaced for each new character entered.

Use a block cursor that appears at the current character position to support overtype mode, as shown in Figure 5.5. This looks the same as the selection of that character and provides the user with a visual cue about the difference between the text-entry modes.

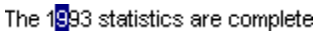



**Figure 5.5 An overtype cursor**

Use the INSERT key to toggle between the normal insert text-entry convention and overtype mode.

## Deleting Text

The DELETE and BACKSPACE keys support deleting text. The DELETE key deletes the character to the right of the text insertion point. BACKSPACE removes the character to the left. In either case, move text in the direction of the deletion to fill the gap — this is sometimes referred to as *auto-joining*. Do not place deleted text on the Clipboard. For this reason, include at least a single-level undo operation in these contexts.

For a text selection, when the user presses DELETE or BACKSPACE, remove the entire block of selected text. Delete text selections when new text is entered directly or by a transfer command. In this case, replace the selected text by the incoming input.

## Handles

Objects may include special control points, called *handles*. You can use handles to facilitate certain types of operations, such as moving, sizing, scaling, cropping, shaping, or auto-filling. The type of handle you use depends on the type of object. For example, the title bar acts as a "move handle" for windows. The borders of the window act as "sizing handles." For icons, the selected icon acts as its own "move handle." In pen-enabled controls, special handles may appear for selection and access to the operations available for an object.

For more information about pen handles, see Pen-Specific Editing Techniques.

A common form of handle is a square box placed at the edge of an object, as shown in Figure 5.6.


**Figure 5.6 A graphic object with handles**

When the handle's interior is solid, the handle implies that it can perform a certain operation, such as sizing, cropping, or scaling. If the handle is "hollow," the handle does not currently support an operation. You can use such an appearance to indicate selection even when an operation is not available.

For more information about the design of handles, see Visual Design.

## Transactions

A *transaction* is a unit of change to an object. The granularity of a transaction may span from the result of a single operation to that of a set of multiple operations. In an ideal model, transactions are applied immediately, and there is support for “rolling back,” or undoing, transactions. Because there are times when this is not practical, specific interface conventions have been established for committing transactions. If there are pending transactions in a window when it is closed, always prompt the user to ask whether to apply or discard the transactions.

Transactions can be committed at different levels, and a commitment made at one level may not imply a permanent change. For example, the user may change font properties of a selection of text, but these text changes may require saving the document file before the changes are permanent.

Use the following commands for committing transactions at the file level.

| Command | Function |
|---|---|
| Save | Saves all interim edits, or checkpoints, to disk and begins a new editing session. |
| Save As | Saves the file (with all interim edits) to a new filename and begins a new editing session. |
| Close | Prompts the user to save any uncommitted edits. If confirmed, the interim edits are saved and the window is removed. |

Use the Save command in contexts where committing file transactions applies to transactions for an entire file, such as a document, and are committed at one time. It may not necessarily apply for transactions committed on an individual basis, such as record-oriented processing.

On a level with finer granularity, you can use the following commands for common handling transactions within a file.

| Command | Function |
|---|---|
| Repeat | Duplicates the last/latest user transaction. |
| Undo | Reverses the last, or specified, transaction. |
| Redo | Restores the most recent, or specified, “undone” transaction. |
| OK | Commits any pending transactions and removes the window. |
| Apply | Commits any pending transactions, but does not remove the window. |
| Cancel | Discards any pending transactions and removes the window. |

Following are the recommended commands for handling process transactions.

| Command | Function |
|---|---|
| Pause | Suspends a process. |
| Resume | Resumes a suspended process. |
| Stop | Halts a process. |

Although you can use the Cancel command to halt a process, Cancel implies that the state will be restored to what it was before the process was initiated.

## Properties

Defining and organizing the properties of an application's components are a key part of evolving toward a more data-centered design. Commands such as Options, Info, Summary Info, and Format often describe features that can be redefined as properties of a particular object (or objects). The Properties command is the common command for accessing the properties of an object; when the user chooses this command, display a secondary window with the properties of the object.

For more information about property sheets, see Secondary Windows.

Defining how to provide access to properties for visible or easily identifiable objects, such as a selection of text, cells, or drawing objects, is straightforward. It may be more difficult to define how to access properties of less tangible objects, such as paragraphs. In some cases, you can include these properties by implication. For example, requesting the properties of a text selection can also provide access to the properties of the paragraph in which the text selection is included.

Another way to provide access to an object's properties is to create a representation of the object. For example, the properties of a page could be accessed through a graphic or other representation of the page in a special area (for example, the status bar) of the window.

Yet another technique to consider is to include specific property entries on the menu of a related object. For example, the pop-up menu of a text selection could include a menu entry for a paragraph. Or consider using the cascading submenu of the Properties command for individual menu entries, but only if the properties are not easily made accessible otherwise. Adding entries for individual properties can easily end up cluttering a menu.

The Properties command is not the exclusive means of providing access to the properties of an object. For example, folder views display certain properties of objects stored in the file system. In addition, you can use toolbar controls to display properties of selected objects.

## Pen-Specific Editing Techniques

A pen is more than just a pointing device. When a standard pen device is installed, the system provides special interfaces and editing techniques.

## Editing in Pen-Enabled Controls

If a pen is installed, the system automatically provides a special interface, called the *writing tool*, to make text editing as easy as possible, enhance recognition accuracy, and streamline correction of errors. The writing tool interface, as shown in Figure 5.7, adds a button to your standard text controls. Because this effectively reduces the visible area of the text box, take this into consideration when designing their size.

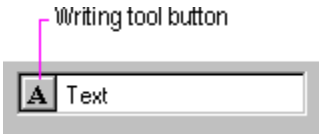



**Figure 5.7 A standard text box with writing tool button**

Figure 5.8 shows how you can also add writing tool support for any special needs of your software, such as a multiline text box.


**Figure 5.8 Adding the writing tool button**

When the text box control has the focus, a selection handle appears, as shown in Figure 5.9. The user can drag this handle to make a selection.

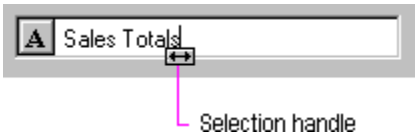



**Figure 5.9 Text box displaying a pen selection handle**

Tapping the writing tool button with a pen (or clicking it with a mouse) presents a special text editing window, as shown in Figure 5.10. Within this window, the user can write text that is recognized automatically.

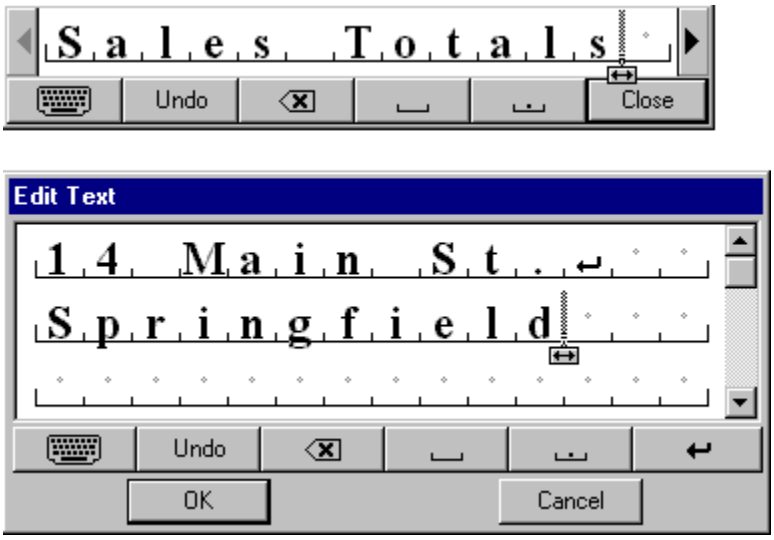



**Figure 5.10 Single and multiline writing tool windows**

In the writing tool editing window, each character is displayed within a special cell. If the user selects text in the original text field, the writing tool window reflects that selection. The user can reset the selection to an insertion point by tapping between characters. This also displays a selection handle that can be dragged to select multiple characters, as shown in Figure 5.11.


**Figure 5.11 Selecting text with the selection handle**

The user can select a single character in its cell by tapping, or double-tapping to select a word. When the user taps a single character, an action handle displays a list of alternative characters, as shown in Figure 5.12.

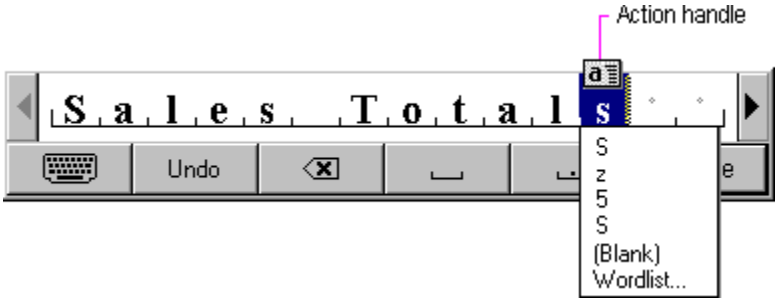



**Figure 5.12 An action handle with a list of alternatives**

Choosing an alternative replaces the selected character and removes the list. Writing over a character or tapping elsewhere also removes the list. The new character replaces the existing one and resets the selection to an insertion point placed to the left of the new character.

The list also includes an item labeled Wordlist. When the user selects this choice, the word that contains the character becomes selected and a list of alternatives is displayed, as shown in Figure 5.13. This list also appears when the user selects a complete word by double-tapping. Choosing an alternative replaces the selected word.

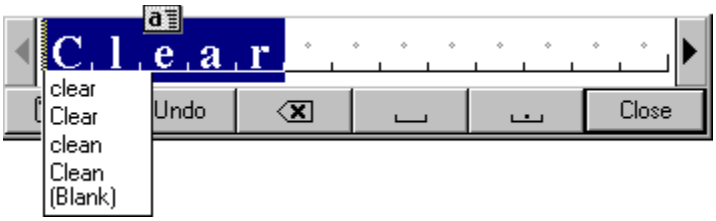



**Figure 5.13 Tapping displays a list of alternatives**

Whenever a selection exists in the window, an action handle appears; the user can use it to perform other operations on the selected items. For example, using the action handle moves or copies the selection by dragging, or the pop-up menu for the selection can be accessed by tapping on the handle, as shown in Figure 5.14.

For more information about pop-up menus, see Menus, Controls, and Toolbars.

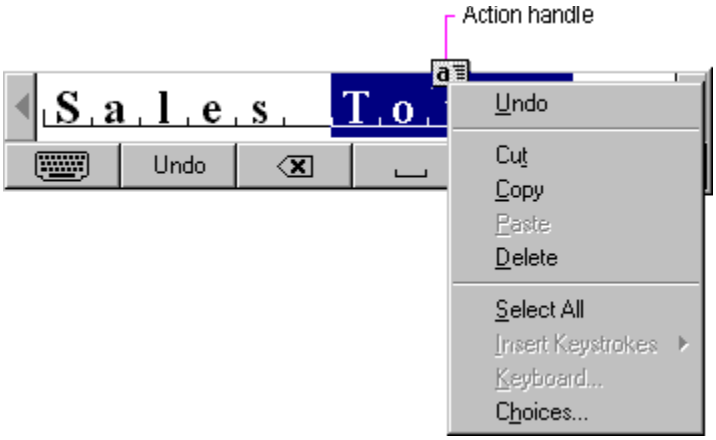



**Figure 5.14 Tapping on the handle displays a pop-up menu**

The buttons on the writing tool window provide for scrolling the text as well as common functions such as Undo, Backspace, Insert Space, Insert Period, and Close (for closing the text window). A multiline writing tool window includes Insert New Line.

The writing tool window also provides a button for access to an onscreen keyboard as an alternative to entering characters with the pen, as shown in Figure 5.15. The user taps the button with the corresponding keyboard glyph on it and the writing tool onscreen keyboard pop-up window replaces the normal writing tool window.

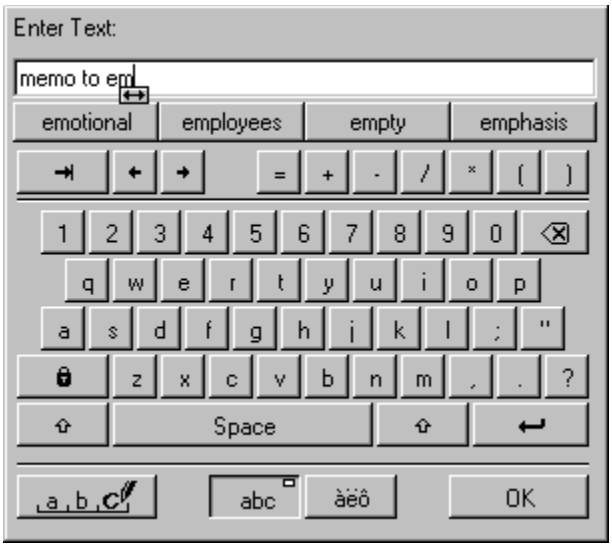



**Figure 5.15 The writing tool onscreen keyboard window**

The writing tool “remembers” its previous use — for text input or as an onscreen keyboard — and opens in the appropriate editing window when subsequently used. In addition, note that when the user displays a writing tool window, it gets the input focus, so avoid using the loss of input focus to a field as an indication that the user is finished with that field or that all text editing occurs directly within a text box.

## Pen Editing Gestures

The pen, when used as a pointing device, supports editing techniques defined for the mouse. When used as a writing device, the pen supports gestures for editing. Gestures (except for Undo) operate positionally, acting upon the objects on which they are drawn. If the user draws a gesture on an unselected object, it applies to that object, even if a selection exists elsewhere within the same selection area. Any pending selections become unselected. If a user draws a gesture over both selected and unselected objects, however, it applies only to the selected ones. If a gesture is drawn over only one element of the selection, it applies to the entire selection. If the gesture is drawn in empty space (on the background), it applies to any existing selection within that selection scope. If no selection exists, the gesture has no effect.

For most gestures, the hot spot of the gesture determines specifically which object the gesture applies to. If the hot spot occurs on any part of a selection, it applies to the whole selection.

Table 5.2 lists the common pen editing gestures. For these gestures, the hot spot of the gesture is the area inside the circle stroke of the gesture.

**Table 5.2 Pen Editing Gestures**

| Gesture | Name | Operation |
|---|---|---|
|  | circled-check | Edit (displays the writing tool editing window) for text; Properties for all other objects. |
|  | circled-c | Copy |
|  | circled-d | Delete (or Clear) |
|  | circled-m | Menu |
|  | circled-n | New line |
|  | circled-p | Paste |
|  | circled-s | Insert space |
|  | circled-t | Insert tab |
|  | circled-u | Undo |
|  | circled-x | Cut |

circled-^ Insert text

These gestures may be localized in certain international versions. In Japanese versions, the circled-k gesture is used to convert Kana to Kanji.

## Transfer Operations

Transfer operations are operations that involve (or can be derived from) moving, copying, and linking objects from one location to another. For example, printing an object is a form of a transfer operation because it can be defined as copying an object to a printer.

Three components make up a transfer operation: the object to be transferred, the destination of the transfer, and the operation to be performed. You can define these components either explicitly or implicitly, depending on which interaction technique you use.

The operation defined by a transfer is determined by the destination. Because a transfer may have several possible interpretations, you can define a default operation and other optimal operations, based on information provided by the source of the transfer and the compatibility and capabilities of the destination. For example, attempting to transfer an object to a container can result in one of the following alternatives:

- Rejecting the object.
- Accepting the object.
- Accepting a subset or transformed form of the object (for example, extract its content or properties but discard its present containment, or convert the object into a new type).

Most transfers are based on one of the following three fundamental operations.

| Operation | Description |
|---|---|
| Move | Relocates or repositions the selected object. Because it does not change the basic identity of an object, a move operation is not the same as copying an object and deleting the original. |
| Copy | Makes a duplicate of an object. The resulting object is independent of its original. Duplication does not always produce an identical clone. Some of the properties of a duplicated object may be different from the original. For example, copying an object may result in a different name or creation date. Similarly, if some component of the object restricts copying, then only the unrestricted elements may be copied. |
| Link | Creates a connection between two objects. The result is usually an object in the destination that provides access to the original. |

There are two different methods for supporting the basic transfer interface: the command method and the direct manipulation method.

## Command Method

The command method for transferring objects uses the Cut, Copy, and Paste commands. Place these commands on the Edit drop-down menu and on the pop-up menu for a selected object. You can also include toolbar buttons to support these commands.

To transfer an object, the user:

1. Makes a selection.
2. Chooses either Cut or Copy.
3. Navigates to the destination (and sets the insertion location, if appropriate).
4. Chooses a Paste operation.

Cut removes the selection and transfers it (or a reference to it) to the Clipboard. Copy duplicates the selection (or a reference to it) and transfers it to the Clipboard. Paste completes the transfer operation. For example, when the user chooses Cut and Paste, remove the selection from the source and relocate it to the destination. For Copy and Paste, insert an independent duplicate of the selection and leave the original unaffected. When the user chooses Copy and Paste Link or Paste Shortcut, insert an object at the destination that is linked to the source.

Choose a form of Paste command that indicates how the object will be transferred into the destination. Use the Paste command by itself to indicate that the object will be transferred as native content. You can also use alternative forms of the Paste command for other possible transfers, using the following general form.

**Paste** [*type name*] [**as** *type name* | **to** *object name*]

For more information about object names, including their type name, see Integrating with the System.

For example, Paste Cells as Word Table, where [*type name*] is Cells and Word Table is the converted type name.

The following summarizes common forms of the Paste command.

| Command | Function |
|---|---|
| Paste | Inserts the object on the Clipboard as native content (data). |
| Paste [*type name*] | Inserts the object on the Clipboard as an OLE embedded object. The OLE embedded object can be activated directly within the destination. |
| Paste [*type name*] as Icon | Inserts the object on the Clipboard as an OLE embedded object. The OLE embedded object is displayed as an icon. |
| Paste Link | Inserts a data link to the object that was copied to the Clipboard. The object's value is integrated or transformed as native content within the destination, but remains linked to the original object so that changes to it are reflected in the destination. |
| Paste Link to [*object name*] | Inserts an OLE linked object, displayed as a picture of the object copied to the Clipboard. The representation is linked to the object copied to the Clipboard so that any changes to the original source object will be reflected in the destination. |
| Paste Shortcut | Inserts an OLE linked object, displayed as a shortcut icon, to the object that was copied to |

| | |
|---|---|
| | the Clipboard. The representation is linked to the object copied to the Clipboard so that any changes to the original source object will be reflected in the destination. |
| Paste Special | Displays a dialog box that gives the user explicit control over how to insert the object on the Clipboard. |

For more information about these Paste command forms and the Paste Special dialog box, see Working with OLE Embedded and OLE Linked Objects.

Use the destination's context to determine what form(s) of the Paste operation to include based on what options it can offer to the user, which in turn may depend on the available forms of the object that its source location object provides. It can also be dependent on the nature or purpose of the destination. For example, a printer defines the context of transfers to it.

Typically, you will need only Paste and Paste Special commands. The Paste command can be dynamically modified to reflect the destination's default or preferred form by inserting the transferred object — for example, as native data or as an OLE embedded object. The Paste Special command can be used to handle any special forms of transfer. Although, if the destination's context makes it reasonable to provide fast access to another specialized form of transfer, such as Paste Link, you can also include that command.

Use the destination's context also to determine the appropriate side effects of the Paste operation. You may also need to consider the type of object being inserted by the Paste operation and the relationship of that type to the destination. The following are some common scenarios:

- When the user pastes into a destination that supports a specific insertion location, replace the selection in the destination with the transferred data. For example, in text or list contexts, where the selection represents a specific insertion location, replace the destination's active selection. In text contexts where there is an insertion location, but there is no existing selection, place the insertion point after the inserted object.
- For destinations with nonordered or Z-ordered contexts where there is no explicit insertion point, add the pasted object and make it the active selection. Also use the destination's context to determine where to place the pasted object. Consider any appropriate user contextual information. For example, if the user chooses the Paste command from a pop-up menu, you can use the pointer's location when the mouse button is clicked to place the incoming object. If the user supplies no contextual clues, place the object at the location that best fits the context of the destination — for example, at the next grid position.
- If the new object is automatically connected (linked) to the active selection (for example, table data and a graph), you may insert the new object in addition to the selection and make the inserted object the new selection.

You also use context to determine whether to display an OLE embedded or OLE linked object as content (view or picture of the object's internal data) or as an icon. For example, you can decide what presentation to display based on what Paste operation the user selects; Paste Shortcut implies pasting an OLE link as an icon. Similarly, the Paste Special command includes options that allow the user to specify how the transferred object should be displayed. If there is no user-supplied preference, the destination application defines the default. For documents, you typically display the inserted OLE object as in its content presentation. If icons better fit the context of your application, make the default Paste operation display the transferred OLE object as an icon.

The execution of a Paste command should not affect the content of the Clipboard. This allows data on the Clipboard to be pasted multiple times, although subsequent Paste operations should always result in copies of the original. However, a subsequent Cut or Copy command replaces the last entry on the Clipboard.

## Direct Manipulation Method

The command method is useful when a transfer operation requires the user to navigate between source and destination. However, for many transfers, direct manipulation is a natural and quick method. In a direct manipulation transfer, the user selects and drags an object to the desired location, but because this method requires motor skills that may be difficult for some users to master, avoid using it as the exclusive transfer method. The best interfaces support the transfer command method for basic operations and direct manipulation transfer as a shortcut technique.

When a pen is being used as a pointing device, or when it drags an action handle, it follows the same conventions as dragging with mouse button 1. For pens with barrel buttons, use the barrel+drag action as the equivalent of dragging with mouse button 2. There is no keyboard interface for direct manipulation transfers.

You can support direct manipulation transfers to any visible object. The object (for example, a window or icon) need not be currently active. For example, the user can drop an object in an inactive window. The drop action activates the window. If an inactive object cannot accept a direct manipulation transfer, it (or its container) should provide feedback to the user.

How the transferred object is integrated and displayed in the drop destination is determined by the destination's context. A dropped object can be incorporated either as native data, an OLE object, a partial form of the object — such as its properties — or a transformed object. You determine whether to add to or replace an existing selection based on the context of the operation, using such factors as the formats available for the object, the destination's purpose, and any user-supplied information such as the specific location that the user drops or commands (or modes) that the user has selected. For example, an application can supply a particular type of tool for copying the properties of objects.

## Default Drag and Drop

*Default drag and drop* transfers an object using mouse button 1. How the operation is interpreted is determined by what the destination defines as the appropriate default operation. As with the command method, the destination determines this based on information about the object (and the formats available for the object) and the context of the destination itself. Avoid defining a destructive operation as the default. When that is unavoidable, display a message box to confirm the intentions of the user.

Using this transfer technique, the user can directly transfer objects between documents defined by your application as well as to system resources, such as folders and printers. Support drag and drop following the same conventions the system supports: the user presses button 1 down on an object, moves the mouse while holding the button down, and then releases the button at the destination. For the pen, the destination is determined by the location where the user lifts the pen tip from the input surface.

The most common default transfer operation is Move, but the destination (dropped on object) can reinterpret the operation to be whatever is most appropriate. Therefore, you can define a default drag and drop operation to be another general transfer operation such as Copy or Link, a destination specific command such as Print or Send To, or even a specialized form of transfer such as Copy Properties.

## Nondefault Drag and Drop

*Nondefault drag and drop* transfers an object using mouse button 2. In this case, rather than executing a default operation, the destination displays a pop-up menu when the user releases the mouse button, as shown in Figure 5.16. The pop-up menu contains the appropriate transfer completion commands

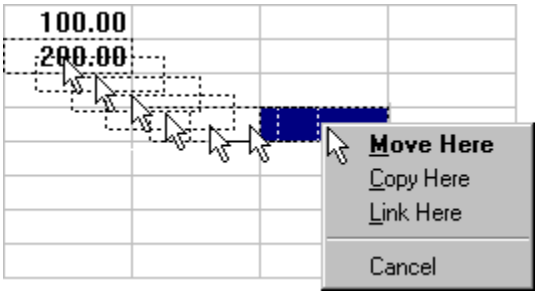



**Figure 5.16 A nondefault drag and drop operation**

The destination always determines which transfer completion commands to include on the resulting pop-up menu, usually factoring in information about the object supplied by the source location.

The form for nondefault drag and drop transfer completion verbs follows similar conventions as the Paste command. Use the common transfer completion verbs, Move Here, Copy Here, and Link Here, when the object being transferred is native data of the destination. When it is not, include the type name. You can also display alternative completion verbs that communicate the context of the destination; for example, a printer displays a Print Here command. For commands that support only a partial aspect or a transformation of an object, use more descriptive indicators — for example, Copy Properties Here, or Transpose Here.

Use the following general form for nondefault drag and drop transfer commands.

[*Command Name*] [*type name* | *object name*] **Here** [**as** *type name*]

The following summarizes common forms for nondefault transfer completion commands.

| **Command** | **Function** |
|---|---|
| Move Here | Moves the selected object to the destination as native content (data). |
| Copy Here | Creates a copy of the selected object in the destination as native content. |
| Link Here | Creates a data link between the selected object and the destination. The original object's value is integrated or transformed as native data within the destination, but remains linked to the original object so that changes to it are reflected in the destination. |
| Move [*type name*] Here<br>Copy [*type name*] Here | Moves or copies the selected object as an OLE embedded object. The OLE embedded object is displayed in its content presentation and can be activated directly within the destination. |
| Link [*type name*] Here | Creates an OLE linked object displayed as a picture of the selected object. The representation is linked to the selected object so that any changes to the original |

| | |
|---|---|
| | object will be reflected in the destination. |
| Move [*type name*] Here as Icon<br>Copy [*type name*] Here as Icon | Moves or copies the selected object as an OLE embedded object and displays it as an icon. |
| Create Shortcut Here | Creates an OLE linked object to the selected object; displayed as a shortcut icon. The representation is linked to the selected object so that any changes to the original object will be reflected in the destination. |

Define and appropriately display one of the commands in the pop-up menu to be the default drag and drop command. This is the command that corresponds to the effect of dragging and dropping with mouse button 1.

For more information about how to display default menu commands, see Visual Design.

## Canceling a Drag and Drop Transfer

When a user drags and drops an object back on itself, interpret the action as cancellation of a direct manipulation transfer. Similarly, cancel the transfer if the user presses the ESC key during a drag transfer. In addition, include a Cancel command in the pop-up menu of a nondefault drag and drop action. When the user chooses this command, cancel the operation.

## Differentiating Transfer and Selection When Dragging

Because dragging performs both selection and transfer operations, provide a convention that allows the user to differentiate between these operations. The convention you use depends on what is most appropriate in the current context of the object, or you can provide specialized handles for selection or transfer. The most common technique uses the location of the pointer at the beginning of the drag operation. If the pointer is within an existing selection, interpret the drag to be a transfer operation. If the drag begins outside of an existing selection, on the background's white space, interpret the drag as a selection operation.

## Scrolling When Transferring by Dragging

When the user drags and drops an object from one scrollable area (such as a window, pane, or list box) to another, some tasks may require transferring the object outside the boundary of the area. Other tasks may involve dragging the object to a location not currently in view. In this latter case, it is convenient to automatically scroll the area (also known as *automatic scrolling* or autoscroll) when the user drags the object to the edge of that scrollable area. You can accommodate both these behaviors by using the velocity of the dragging action. For example, if the user is dragging the object slowly at the edge of the scrollable area, you scroll the area; if the object is being dragged quickly, do not scroll.

To support this technique during a drag operation, you sample the pointer's position at the beginning of the drag each time the mouse moves, and on an application-set timer (every 100 milliseconds recommended). If you use OLE drag and drop support, you need not set a timer. Store each value in an array large enough to hold at least three samples, replacing existing samples with later ones. Then calculate the pointer's velocity based on at least the last two locations of the pointer.

To calculate the velocity, sum the distance between the points in each adjacent sample and divide the total by the sum of the time elapsed between samples. Distance is the absolute value of the difference between the x and y locations, or (abs(x1 - x2) + abs(y1 - y2)). Multiply this by 1024 and divide it by the elapsed time to produce the velocity. The 1024 multiplier prevents the loss of accuracy caused by integer division.

Distance as implemented in this algorithm is not true Cartesian distance. This implementation uses an approximation for purposes of efficiency, rather than using the square root of the sum of the squares, (sqrt((x1 - x2)^2 + (y1 - y2)^2)), which is more computationally expensive.

You also predefine a hot zone along the edges of the scrollable area and a scroll time-out value. Use twice the width of a vertical scroll bar or height of a horizontal scroll bar to determine the width of the hot zone.

During the drag operation, scroll the area if the following conditions are met: the user moves the pointer within the hot zone, the current velocity is below a certain threshold velocity, and the scrollable area is able to scroll in the direction associated with the hot zone it is in. The recommended threshold velocity is 20 pixels per second. These conventions are illustrated in Figure 5.17.

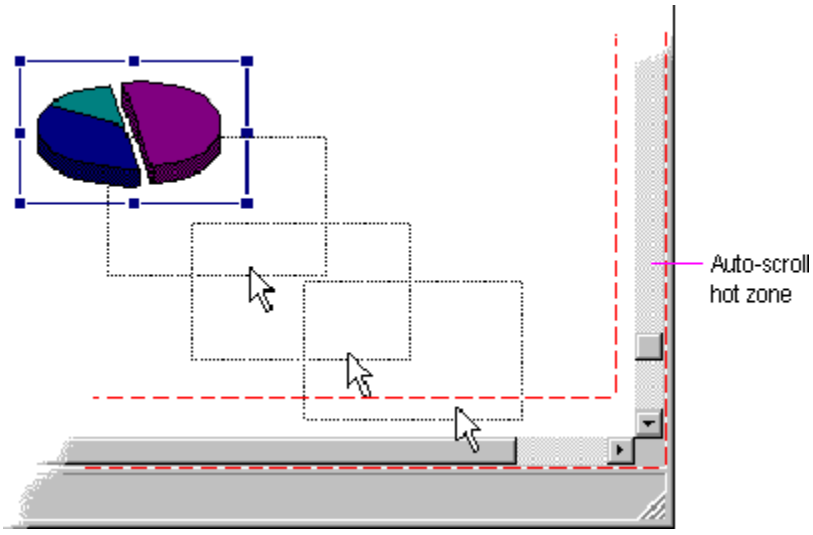



**Figure 5.17 Automatic scrolling based on velocity of dragging**

The amount you scroll depends on the type of information and reasonable scrolling distance. For

example, for text, you typically scroll vertically one line at a time. Consider using the same scrolling granularity that is provided for the scroll bar arrows.

For more information about scrolling, see Windows.

To support continuous scrolling, determine what the scroll frequency you want to support — for example, four lines per second. After using a velocity check to initiate auto-scrolling, set a timer — for example, 100 milliseconds. When the timer expires, determine how long it has been since the last time you scrolled. If the elapsed time is greater than your scrolling frequency, scroll another unit. If not, reset your timer and check again when the timer completes.

## Transfer Feedback

Because transferring objects is one of the most common user tasks, providing appropriate feedback is an important design factor. Inconsistent or insufficient feedback can result in user confusion.

For more information about designing transfer feedback, see Visual Design.

## Command Method Transfers

For a command method transfer, remove the selected object visually when the user chooses the Cut command. If there are special circumstances that make removing the object's appearance impractical, you can instead display the selected object with a special appearance to inform the user that the Cut command was completed, but that the object's transfer is pending. For example, the system displays icons in a checkerboard dither to indicate this state. You will also need to restore the visual state of the object if the user chooses Cut or Copy for another object before choosing a Paste command, effectively canceling the pending Cut command. The user will expect Cut to remove a selected object, so carefully consider the impact of inconsistency if you choose this alternate feedback.

The Copy command requires no special feedback. A Paste operation also requires no further feedback than that already provided by the insertion of the transferred object. However, if you did not remove the display of the object and used an alternate representation when the user chose the Cut command, you must remove it now.

## Direct Manipulation Transfers

During a direct manipulation transfer operation, provide visual feedback for the object, the pointer, and the destination. Specifically:

- Display the object with selected appearance while the view it appears in has the focus. To indicate that the object is in a transfer state, you can optionally display the object with some additional appearance characteristics. For example, for a move operation, you can use the checkerboard dithered appearance used by the system to indicate when an icon is Cut. Change this visual state based on the default completion operation supported by the destination the pointer is currently over. Retain the representation of the object at the original location until the user completes the transfer operation. This not only provides a visual cue to the nature of the transfer operation, it provides a convenient visual reference point.
- Display a representation of the object that moves with the pointer. Use a presentation that provides the user with information about how the information will appear in the destination and that does not obscure the context of the insertion location. For example, when transferring an object into a text context, it is important that the insertion point not be obscured during the drag operation. A translucent or outline representation, as shown in Figure 5.18, works well because it allows the underlying insertion location to be seen while also providing information about the size, position, and nature of the object being dragged.

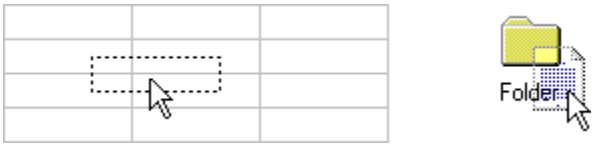



**Figure 5.18 Outline and translucent representations for transfer operations**

- The object's existing source location provides the transferred object's initial appearance, but any destination can change the appearance. Design the presentation of the object to provide feedback as to how the object will be integrated by that destination. For example, if an object will be embedded as an icon, display the object as an icon. If the object will be incorporated as part of the native content of the destination, then the presentation of the object that the destination displays should reflect that. For example, if a table being dragged into a document will be incorporated as a table, the representation could be an outline or translucent form of the table. On the other hand, if the table will be converted to text, display the table as a representation of text, such as a translucent presentation of the first few words in the table.
- Display the pointer appropriate to the context of the destination, usually used for inserting objects. For example, when dragging an object into a text editing context such that the object will be inserted between characters, display the usual text editing pointer (sometimes called the I-beam pointer).
- Display the interpretation of the transfer operation at the lower right corner of the pointer, as shown in Figure 5.19. No additional glyph is required for a move operation. Use a plus sign (+) when the transfer is a copy operation. Use the shortcut arrow graphic for linking.

**Figure 5.19 Pointers - move, copy, and link operations**

- Use visual feedback to indicate the receptivity of potential destinations. You can use selection highlighting and optionally animate or display a representation of the transfer object in the destination. Optionally, you can also indicate when a destination cannot accept an object by using the "no drop" pointer when the pointer is over it, as shown in Figure 5.20.

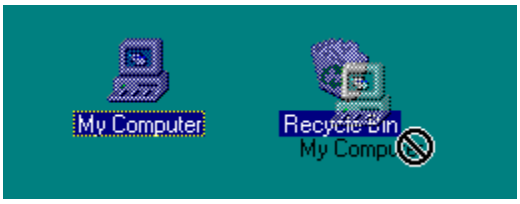



**Figure 5.20 A “no drop “ pointer**

## Specialized Transfer Commands

In some contexts, a particular form of a transfer operation may be so common, that introducing an additional specialized command is appropriate. For example, if copying existing objects is a frequent operation, you can include a Duplicate command. Following are some common specialized transfer commands.

| Command | Function |
|---|---|
| Delete | Removes an object from its container. If the object is a file, the object is transferred to the Recycle Bin. |
| Clear | Removes the content of a container. |
| Duplicate | Copies the selected object. |
| Print | Prints the selected object on the default printer. |
| Send To | Displays a list of possible transfer destinations and transfers the selected object to the user selected destination. |

Delete and Clear are often used synonymously. However, they are best differentiated by applying Delete to an object and Clear to the container of an object.

## Shortcut Keys for Transfer Operations

Following are the defined shortcut techniques for transfer operations.

| Shortcut | Operation |
|---|---|
| CTRL+X | Performs a Cut command. |
| CTRL+C | Performs a Copy command. |
| CTRL+V | Performs a Paste command. |
| CTRL+drag | Toggles the meaning of the default direct manipulation transfer operation to be a copy operation (provided the destination can support the copy operation). The modifier may be used with either mouse button. |
| ESC | Cancels a drag and drop transfer operation. |

For more information about reserved and recommended shortcut key assignments, see Keyboard Interface Summary.

Because of the wide use of these command shortcut keys throughout the interface, do not reassign them to other commands.

## Creation Operations

Creating new objects is a common user action in the interface. Although applications can provide the context for object creation, avoid considering an application's interface as the exclusive means of creating new objects. Creation is typically based on some predefined object or specification and can be supported in the interface in a number of ways.

## Copy Command

Making a copy of an existing object is the fundamental paradigm for creating new objects. Copied objects can be modified and serve as prototypes for the creation of other new objects. The transfer model conventions define the basic interaction techniques for copying objects. Copy and Paste commands and drag and drop manipulation provide this interface.

## New Command

The New command facilitates the creation of new objects. New is a command applied to a specific object, automatically creating a new instance of the object's type. The New command differs from the Copy and Paste commands in that it is a single command that generates a new object.

## Insert Command

The Insert command works similarly to the New command, except that it is applied to a container to create a new object, usually of a specified type, in that container. In addition to inserting native types of data, use the Insert command to insert objects of different types. By supporting OLE, you can support the creation of a wide range of objects. In addition, objects supported by your application can be inserted into data files created by other OLE applications.

For more information about inserting objects, see Working with OLE Embedded and OLE Linked Objects.

## Using Controls

You can use controls to support the automatic creation of new objects. For example, in a drawing application, buttons are often used to specify tools or modes for the creation of new objects, such as drawing particular shapes or controls. Buttons can also be used to insert OLE objects.

For more information about using buttons to create new objects, see Working with OLE Embedded and OLE Linked Objects.

## Using Templates

A *template* is an object that automates the creation of a new object. To distinguish its purpose, display a template icon as a pad with the small icon of the type of the object to be created, as shown in Figure 5.21.

**Figure 5.21 A template icon**

Define the New command as the default operation for a template object; this starts the creation process, which may either be automatic or request specific input from the user. Place the newly created object in the same location as the container of the template. If circumstances make that impractical, place the object in a common location, such as the desktop, or, during the creation process, include a prompt that allows a user to specify some other destination. In the former situation, display a message box informing the user where the object will appear.

## Operations on Linked Objects

A *link* is a connection between two objects that represents or provides access to another object that is in another location in the same container or in a different, separate container. The components of this relationship include the link source (sometimes referred to as the referent) and the link or linked object (sometimes referred to as the reference). A linked object often has operations and properties independent of its source. For example, a linked object's properties can include attributes like update frequency, the path description of its link source, and the appearance of the linked object. The containers in which they reside provide access to and presentation of commands and properties of linked objects.

Links can be presented in various ways in the interface. For example, a *data link* propagates a value between two objects, such as between two cells in a worksheet or a series of data in a table and a chart. *Jumps* (also referred to as hyperlinks) provide navigational access to another object. An *OLE linked object* provides access to any operation available for its link source and also supplies a presentation of the link source. A shortcut icon is a link, displayed as an icon.

For more information about OLE linked objects, see Working with OLE Embedded and OLE Linked Objects. For more information about jumps, see User Assistance.

When the user transfers a linked object, store both the absolute and relative path to its link source. The absolute path is the precise description of its location, beginning at the root of its hierarchy. The relative path is the description of its location relative to its current container.

The destination of a transfer determines whether to use the absolute or relative path when the user accesses the link source through the linked object. The relative path is the most common default path. However, regardless of which path you use, if it fails, use the alternative path. For example, if the user copies a linked object and its link source to another location, the result is a duplicate of the linked object and the link source. The relative path for the duplicate linked object is the location of the duplicate of the link source. The absolute path for the duplicate linked object is the description of the location of the initial link source. Therefore, when the user accesses the duplicate of the linked object, its inferred connection should be with the duplicate of the link source. If that connection fails — for example, because the user deletes the duplicate of the linked source — use the absolute path, the connection to the original link source.

Optionally, you can make the preferred path for a linked object a field in the property sheet for linked object. This allows the user to choose whether to have a linked object make use of the absolute or relative path to its link source.

When the user applies a link operation to a linked object, link to the linked object rather than its linked source. That is, linking a linked object results in a linked object linked to a linked object. If such an operation is not valid or appropriate - for example, because the linked object provides no meaningful context - then disable any link commands or options when the user selects a linked object.

Activation of a linked object depends on the kind of link. For example, a single click can activates a jump. However, a single click only results in selecting a data link or an OLE linked object. If you use a single click to do anything other than select the linked object, distinguish the object by either presenting it as a button control, displaying the hand pointer (as shown in Figure 5.22) when the user moves the pointer over the linked object, or both. These techniques provide feedback to the user that the clicking involves more than selection.

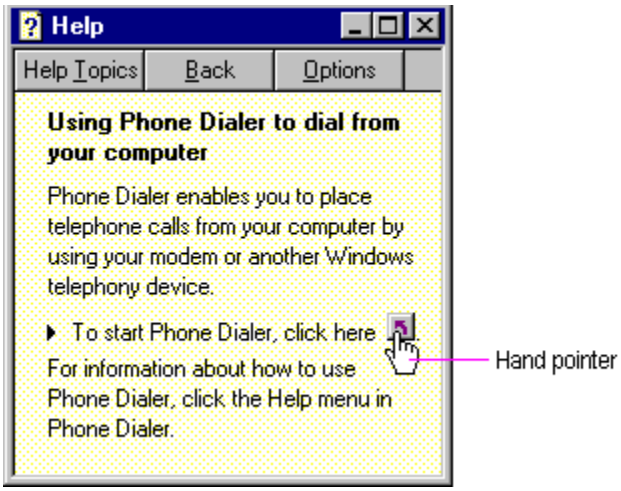



**Figure 5.22 The hand pointer**

## Introduction

Windows provide the fundamental way a user views and interacts with data. Consistency in window design is particularly important because it enables users to easily transfer their learning skills and focus on their tasks rather than learn new conventions. This topic describes the common window types and presents guidelines for general appearance and operation.

## Common Types of Windows

Because windows provide access to different types of information, they are classified according to common usage. Interacting with objects typically involves a *primary window* in which most primary viewing and editing activity takes place. In addition, multiple supplemental *secondary windows* can be included to allow users to specify parameters or options, or to provide more specific details about the objects or actions included in the primary window.

For more information about secondary windows, see Secondary Windows.

## Primary Window Components

A typical primary window consists of a frame (or border) which defines its extent, and a title bar which identifies what is being viewed in the window. If the viewable content of the window exceeds the current size of the window, scroll bars are used. The window can also include other components like menu bars, toolbars, and status bars.

Figure 6.1 shows the common components of a primary window.

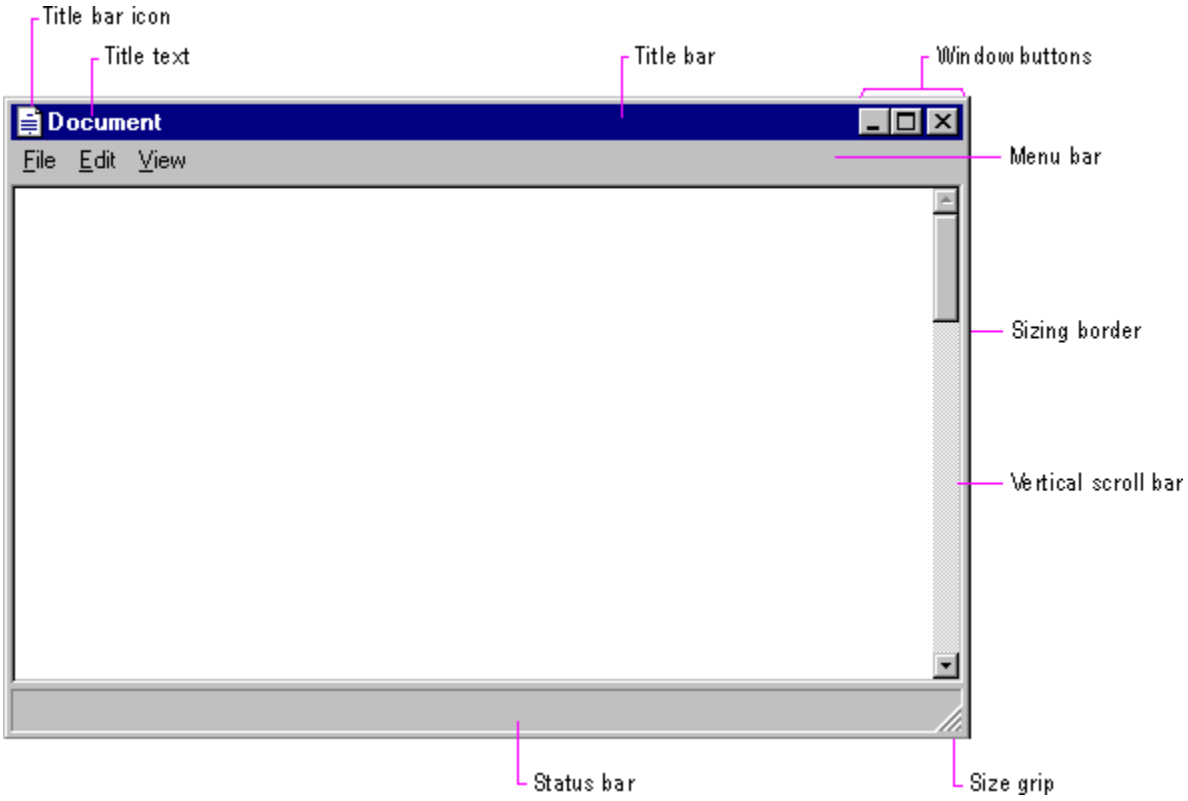



**Figure 6.1 A primary window**

## Window Frames

Every window has a boundary that defines its shape. A sizable window has a distinct border that provides control points (handles) for resizing the window using direct manipulation. If the window cannot be resized, the border coincides with the edge of the window.

## Title Bars

At the top edge of the window, inside its border, is the *title bar* (also referred to as the caption or caption bar), which extends across the width of the window. The title bar identifies what the window is viewing. It also serves as a control point for moving the window and an access point for commands that apply to the window and its associated view. For example, clicking on the title bar with mouse button 2 displays the pop-up menu for the window. Pressing the ALT+SPACEBAR key combination also displays the pop-up menu for the window.

## Title Bar Icons

A primary window includes the small version of the object's icon. The small icon appears in the upper left corner of the title bar and represents the object being viewed in the window. If the window represents a "tool" or utility application (that is, an application that does not create, load, and save its own data files), use the small version of the application's icon in its title bar, as shown in Figure 6.2.

For information about how to register icons for your application and data file types, see Integrating with the System. For more information about designing icons, see Visual Design.

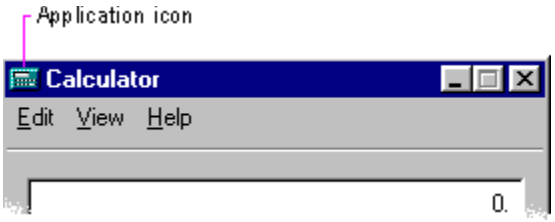



**Figure 6.2 "Tool" title bar**

If the application creates, loads and saves documents or data files and the window represents the view of one of its files, use the icon that represents its document or data file type in the title bar, as shown in Figure 6.3. Display the data file icon even if the user has not saved the file yet, rather than displaying the application icon and then the data file icon once the user saves the file.

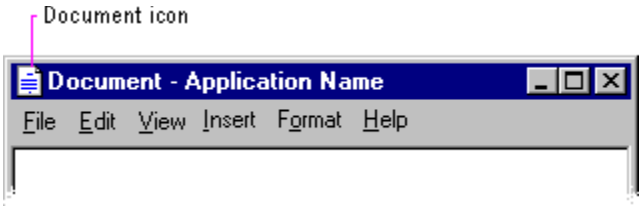



**Figure 6.3 Document title bar**

If an application uses the multiple document interface (MDI) design, place the application's icon in the parent window's title bar, and place an icon that reflects the application's data file type in the child window's title bar, as shown in Figure 6.4.

For more information about MDI, see Window Management.

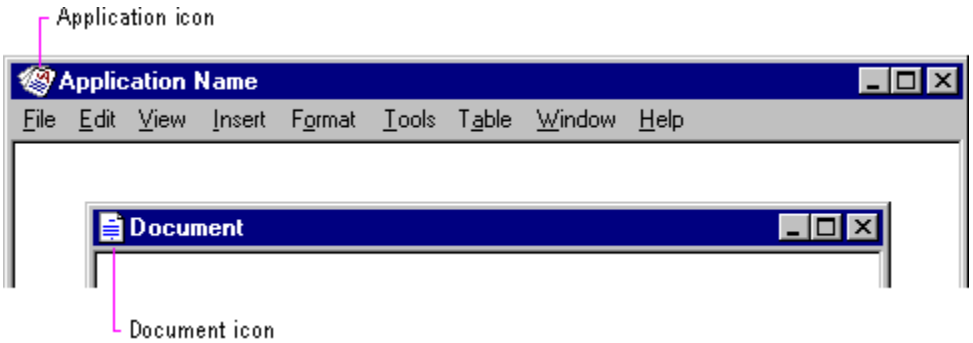



**Figure 6.4 MDI application and document title bars**

However, when a user maximizes the child window, and you hide its title bar and merge its title information with the parent, display the icon from the child window's title bar in the menu bar of the parent window. If multiple child windows are open within the MDI parent window, display only the icon from the active (topmost) child window.

When the user clicks the title bar icon with mouse button 2, display the pop-up menu for the object. Typically, the menu contains a similar set of commands that are available for the icon from which the window was opened, except that Close replaces Open. Also define Close as the default command, so when the user double-clicks the title bar icon, the window closes. Clicking elsewhere with button 2 on the title bar displays the pop-up menu for the window.

When the user clicks the title bar icon with mouse button 1, the system also displays the pop-up menu for the window. However, this behavior is only supported for backward compatibility with Windows 3.1. Avoid documenting it as the primary way to access the pop-up menu for the window. Instead, document the use of button 2 as the correct way to display the pop-up menu for the window.

## Title Text

The window title text identifies the name of the object being viewed in the window. It should always correspond to the icon of the type you display in the title bar. It should also match the label of the icon in the file system that represents the object. For example, if the user opens a data file named “My Document” in the resulting window, you display the icon for that document type followed by the name of the data file. You may also include the name of the application in use; however, if it is used, display the name of the data file first, followed by a dash and the application name, as shown in Figure 6.5.

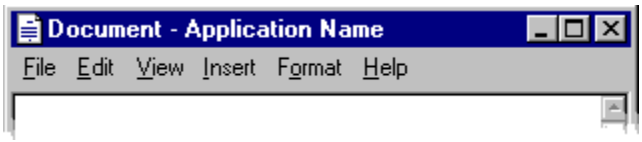



**Figure 6.5 Title text order: document name — application name**

The order of the document (or data) filename and application name differs from the Windows 3.1 guidelines. The new convention is better suited for the design of a data-centered interface.

If the window represents a “tool” application that does not create or edit its own data files, such as the Windows Calculator, display the application’s name, as displayed for the application’s icon label, in the title bar. If the tool application operates as a utility for other files created by other applications — such as a special viewer or browser application — where the view displayed is not the primary open view of the file, or where the “tool” application requires an additional specification to indicate its context — such as the Windows Explorer — place the name of the application first, then include a dash and the specification text. For example, the title text of the Windows Explorer includes the name of the current container displayed in the browser.

For an MDI application, use the application’s name in the parent window and the data file’s name in the child windows. When the user maximizes the file’s child window, format the title text following the same convention as a tool application, with the application’s name first, followed by the data filename, as shown in Figure 6.6.

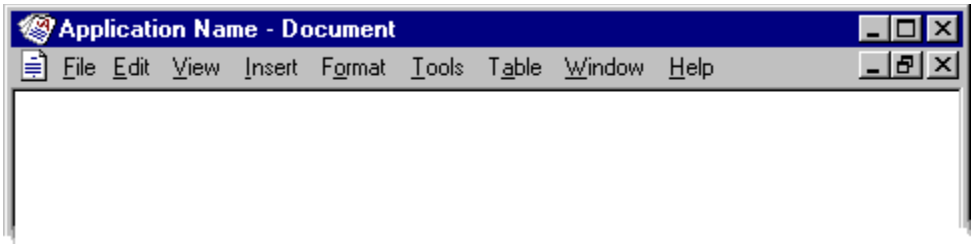



**Figure 6.6 Document follows application name for maximized child window**

When the user directly opens an application that displays a new data file, supply a name for the file and place it in the title bar, even if the user has not saved the file yet. Use the type name — for example Document (*n*), Sheet (*n*), Chart (*n*), where *n* is a number, as in Document (1). Make certain that the proposed name does not conflict with an existing name in the current directory. Also use this name as the proposed default filename for the object in the Save As dialog box. If it is impractical or inappropriate to supply a default name, display a placeholder in the title, such as (Untitled).

For more information about type names, see Integrating with the System. For more information about the Save As dialog box, see Secondary Windows.

Follow the same convention if your application includes a New command that creates new files. Avoid

prompting the user for a name. Instead, you can supply a Save As dialog box that allows the user to confirm or change your proposed name when they save or close the file or attempt to create a new file.

Display a filename in the title bar exactly as it appears to the user in the file system, using both uppercase and lowercase letters. However, avoid displaying the file's extension or the path in the title bar. This information is not meaningful for most users and can make it more difficult for them to identify the file. However, because the system provides an option for users to display filename extensions, use the system-supplied functions to format a filename, which will display the filename appropriately based on the user's preference.

The **GetFileTitle** and **SHGetFileInfo** functions automatically format names correctly. For more information about these functions, see the documentation included in the Microsoft Win32 Software Development Kit (SDK).

If your application supports multiple windows for viewing the same file, you may use the title text to distinguish between the views — but use a convention that will not be confused as part of the filename. For example, you may want to append :*n*, where *n* represents the instance of the window, as in Document:2. Make certain you do not include this view designation as part of the filename you supply in the Save As dialog box.

If the name of the displayed object in the window changes — for example, when the user edits the name in the object's property sheet — update the title text to reflect that change. Always try to maintain a clear association between an object and its open window.

The title text and title bar icon should always represent the outmost container — the object that was opened — even if the user selects an embedded object or navigates the internal hierarchy of the object being viewed in the window. If you need an additional specification to clarify what the user is viewing, place this specification after the filename and separate it clearly from the filename, such as enclosing it in parentheses — for example, My HardDisk (C:). Because the system now supports long filenames, avoid additional specification whenever possible. Complex or verbose additions to the title text also make it more difficult for the user to easily read and identify the window.

When the width of the window does not allow you to display the complete title text, you may abbreviate the title text, being careful to maintain the essential information that allows the user to quickly identify the window.

For more information about abbreviating names, see Integrating with the System.

Avoid drawing directly into the title bar or adding other controls. Such added items can make reading the name in the title difficult, particularly because the size of the title bar varies with the size of the window. In addition, the system uses this area for displaying special controls. For example, in some international versions of Windows, the title area provides information or controls associated with the input of certain languages.

## Title Bar Buttons

Include command buttons associated with the common commands of the primary window in the title bar. These act as shortcuts to specific window commands. Clicking a title bar button with mouse button 1 invokes the command associated with the command button. Optionally, you can also support clicking a title bar command button with mouse button 2 to display the pop-up menu for the window. For the pen, tapping a window button invokes its associated command, and optionally you may support barrel-tapping it (or using the pen menu gesture) to display the pop-up menu for the window.

In a typical situation, one or more of the following buttons appear in a primary window (provided that the window supports the respective functions).

**6.1 Title Bar Buttons**

| Button | Command | Operation |
|---|---|---|
|  | Close | Closes the window. |
|  | Minimize | Minimizes the window. |
|  | Maximize | Maximizes the window. |
|  | Restore | Restores the window. |

The system does not support the inclusion of the context-sensitive Help button available for secondary windows. Applications wishing to provide this functionality can do so by including a Help toolbar button. Similarly, avoid including Maximize, Minimize, and Restore buttons in the title bars of secondary windows because those commands do not apply to those windows.

When displaying these buttons, use the following guidelines:

- When a command is not supported by a window, do not display its corresponding button.
- The Close button always appears as the rightmost button. Leave a gap between it and any other buttons.
- The Minimize button always precedes the Maximize button.
- The Restore button always replaces the Maximize button or the Minimize button when that command is carried out.

## Basic Window Operations

The basic operations for a window include: activation and deactivation, opening and closing, moving, sizing, scrolling, and splitting. The following sections describe these operations.

## Activating and Deactivating Windows

While the system supports the display of multiple windows, the user generally works within a single window at a time. This window is called the *active window*. The active window is typically at the top of the window Z order. It is also visually distinguished by its title bar that is displayed in the active window title color. All other windows are *inactive* with respect to the user's input; that is, while other windows can have ongoing processes, only the active window receives the user's input. The title bar of an inactive window displays the system inactive window color. Your application can query the system for the current values for the active title bar color and the inactive title bar color.

For more information about using the **GetSysColor** function to access the COLOR_ACTIVECAPTION and COLOR_INACTIVECAPTION constants, see the documentation included in the Win32 SDK.

The user activates a primary window by switching to it; this inactivates any other primary windows. To activate a window with the mouse or pen, the user clicks or taps on any part of the window, including its interior. If the window is minimized, the user clicks (taps) the button representing the window in the taskbar. From the keyboard, the system provides the ALT+TAB key combination for switching between primary windows. The SHIFT+ALT+TAB key also switches between windows, but in reverse order. (The system also supports ALT+ESC for switching between windows.) The reactivation of a window should not affect any pre-existing selection within it; the selection and focus are restored to the previously active state.

When the user reactivates a primary window, the window and all its secondary windows come to the top of the window order and maintain their relative positions. If the user activates a secondary window, its primary window comes to the top of the window order along with the primary window's other secondary windows.

When a window becomes inactive, hide the selection feedback (for example, display of highlighting or handles) of any selection within it to prevent confusion over which window is receiving keyboard input. A direct manipulation transfer (drag and drop) is an exception. Here, you can display transfer feedback if the pointer is over the window during the drag operation. Do not activate the window unless the user releases the mouse button (the pen tip is lifted) in that window.

## Opening and Closing Windows

When the user opens a primary window, include an entry for it on the taskbar. If the window has been opened previously, restore the window to its size and position when it was last closed. If possible and appropriate, reinstate the other related view information, such as selection state, scroll position, and type of view. When opening a primary window for the first time, open it to a reasonable default size and position as best defined by the object or application. For details about storing state information in the system registry, see Integrating with the System.

Only primary windows, not secondary windows, should include an entry on the taskbar.

Because display resolution and orientation varies, your software should not assume a fixed display size, but rather adapt to the shape and size defined by the system. If you use standard system interfaces the system automatically places your windows relative to the current display configuration.

The **SetWindowPlacement** function is an example of a system interface that will automatically place windows correctly relative to the current display. For more information about this function, see the documentation included in the Win32 SDK.

Opening the primary window activates that window and places it at the top of the window order. If the use attempts to open a primary window that is already open within the same desktop, activate the existing window using the following recommendations. If the existing window is minimized, restore it when you activate it.

| File type | Action when repeating an Open operation |
|---|---|
| Document or data file | Activates the existing window of the object and displays it at the top of the window Z order. |
| Application file | Displays a message box indicating that an open window of that application already exists and offers the user the option to switch to the open window or to open another window. Either choice activates the selected window and brings it to the top of the window Z order. |
| Document file that is already open in an MDI application window | Activates the existing window of the file. Its MDI parent window comes to the top of the window Z order, and the file appears at the top of the Z order within its MDI parent window. |
| Document file that is not already open, but its associated MDI application is already running (open) | Opens a new instance of the file's associated MDI application at the top of the window Z order and displays the child window for the file. Optionally, as an alternative, displays a message box indicating that an open window of that application already exists and offers the user the option to use the existing window or to open a new parent window. |

For more information about MDI, see Window Management.

The user closes a primary window by clicking (for a pen, tapping the screen) the Close button in the title bar or choosing the Close command from the window's pop-up menu. Although the system supports double-clicking (with a pen, double-tapping the screen) on the title bar icon as a shortcut for closing the window for compatibility with previous versions of Windows, avoid documenting this as the primary way to close a primary window. Instead, document the Close button.

When the user chooses the Close command, if your application does not automatically save these changes and pending transactions or edits that have not yet been saved to file remain, display a message asking the user whether to save any changes, discard any changes, or cancel the Close operation before closing the window. If there are no pending transactions, just close the window. Follow this same convention for any other command that results in closing the primary window (for example, Exit or Shut Down).

For more information about supporting the Close command, see General Interaction Techniques.

When closing the primary window, close any of its dependent secondary windows as well. The design of your application determines whether closing the primary window also ends the application processes. For example, closing the window of a text document typically halts any application code or processes remaining for inputting or formatting text. However, closing the window of a printer has no effect on the jobs in the printer's queue. In both cases, closing the window removes its entry from the taskbar.

## Moving Windows

The user can move a window either by dragging its title bar using the mouse or pen or by using the Move command on the window's pop-up menu. On most configurations, an outline representation moves with the pointer during the operation, and the window is redisplayed in the new location after the completion of the move. (The system also provides a display property setting that redraws the window dynamically as it is moved.) After choosing the Move command, the user can move the window with the keyboard interface by using arrow keys and pressing the ENTER key to end the operation and establish the window's new location. Never allow the user to reposition a window so that it cannot be accessed.

A window need not be active before the user can move it. The action of moving the window implicitly activates it.

Moving a window may clip or reveal information shown in the window. In addition, activation can affect the view state of the window — for example, the current selection can be displayed. However, when the user moves a window, avoid making any changes to the content being viewed in that window.

# Resizing Windows

Make your primary windows resizable unless the information displayed in the window is fixed, such as in the Windows Calculator program. The system provides several conventions that support user resizing of a window.

## Sizing Borders

The user resizes a primary window by dragging the sizing border with the mouse or pen at the edge of a window or by using the Size command on the window's menu. An outline representation of the window moves with the pointer. (On some configurations, the system may include a display option to dynamically redraw the window as it is sized.) After completing the size operation, the window assumes its new size. Using the keyboard, the user can size the window by choosing the Size command, using the arrow keys, and pressing the ENTER key.

A window does not need to be active before the user can resize it. The action of sizing the window implicitly makes it active, and it remains active after the sizing operation.

When the user resizes a window to be smaller, you must determine how to display the information viewed in that window. Use the context and type of information to help you choose your approach. The most common approach is to clip the information. However, in other situations where you want the user to see as much information as possible, you may want to consider using different methods, such as rewrapping or scaling the information. Use these variations carefully because they may not be consistent with the resizing behavior of most windows. In addition, avoid these methods when readability or maintaining the structural relationship of the information is important.

Although the size of a primary window can vary, based on the user's preference, you can define a window's maximum size. When defining this size, consider the reasonable usage within the window, and the size and orientation of the screen.

# Maximizing Windows

Although the user may be able to directly resize a window to its maximum size, the Maximize command optimizes this operation. Include this command on a window's pop-up menu, and as the Maximize command button in the title bar of the window.

Maximizing a window increases the size of the window to its largest, optimum size. The system default setting for the maximum size is as large as the display, excluding the space used by the taskbar (or other application-defined desktop toolbars). For an MDI child window, the default maximize size fills the parent window. But, you can define the size to be less or, in some cases, more than the default dimensions.

When the user maximizes a window, replace the Maximize button with a Restore button. Then, disable the Maximize command and enable the Restore command on the pop-up menu for the window.

# Minimizing Windows

Minimizing a window reduces it to its smallest size. To support this command, include it on the pop-up menu for the window and as the Minimize command button in the title bar of the window.

For primary windows, minimizing removes the window from the screen, but leaves its entry in the taskbar. For MDI child windows, the window resizes to a minimum size within its parent window. To minimize a window, the user chooses the Minimize command from the window's pop-up menu or the Minimize command button on the title bar.

Because the representation of minimized windows changed in Microsoft Windows 95, using an icon as the way to reflect state information may not be appropriate. As an alternative, you may want to consider creating a status indicator for the taskbar. For more information about status notification, see Integrating with the System.

When the user minimizes a window, disable the Minimize command on the pop-up menu and enable the Restore command.

## Restoring Windows

Support the Restore command to restore a window to its previous size and position after the user has maximized or minimized the window. For maximized windows, enable this command on the window's pop-up menu and replace the Maximize button with the Restore button in the title bar of the window.

For minimized windows, also enable the Restore command in the pop-up menu of the window. The user restores a minimized primary window to its former size and position by clicking (for pens, tapping the screen) on its button in the taskbar that represents the window, selecting the Restore command on the pop-up menu of the window's taskbar button, or using the ALT+TAB (or the SHIFT+ALT+TAB) key combination.

## Size Grip

When you define a sizable window, you may include a size grip. A *size grip* is a special handle for sizing a window. It is not exclusive to the sizing border. To size the window, the user drags the grip and the window resizes following the same conventions as the sizing border.

Always locate the size grip in the lower right corner of the window. Typically, this means you place the size grip at the right end of a horizontal scroll bar or the bottom of a vertical scroll bar. However, if you include a status bar in the window, display the size grip at the far corner of the status bar instead. Never display the size grip in both locations at the same time.

For more information about the use of the size grip in a status bar, see Menus, Controls, and Toolbars.

## Scrolling Windows

When the information viewed in a window exceeds the size of that window, the window should support scrolling. Scrolling enables the user to view portions of the object that are not currently visible in a window. Scrolling is commonly supported through the use of a scroll bar. A *scroll bar* is a rectangular control consisting of *scroll arrows*, a *scroll box*, and a *scroll bar shaft*, as shown in Figure 6.7.

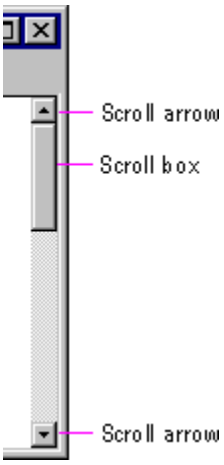



**Figure 6.7 Scroll bar and its components**

You can include a vertical scroll bar, a horizontal scroll bar, or both. Align a scroll bar with the vertical or horizontal edge of the window orientation it supports. If the content is never scrollable in a particular direction, do not include a scroll bar for that direction.

Scroll bars are also available as separate window components. For more information about scroll bar controls, see Menus, Controls, and Toolbars.

The common practice is to display scroll bars if the view requires some scrolling under any circumstances. If the window becomes inactive or resized so that its content does not require scrolling, you should continue to display the scroll bars. While removing the scroll bars when the window is inactive potentially allows the display of more information and feedback about the state of the window, it also requires the user to explicitly activate the window to scroll. Consistently displaying scroll bars provides a more stable environment.

## Scroll Arrows

Scroll arrow buttons appear at each end of a scroll bar, pointing in opposite directions away from the center of the scroll bar. The scroll arrows point in the direction that the window “moves” over the data. When the user clicks (for pens, tapping the screen) a scroll arrow, the data in the window moves, revealing information in the direction of the arrow in appropriate increments. The granularity of the increment depends on the nature of the content and context, but it is typically based on the size of a standard element. For example, you can use one line of text for vertical scrolling, one row for spreadsheets. You can also use an increment based a fixed unit of measure. Whichever convention you choose, maintain the same scrolling increment throughout a window. The objective is to provide an increment that provides smooth but efficient scrolling. When a window cannot be scrolled any further in a particular direction, disable the scroll arrow corresponding to that direction.

The default system support for scroll bars does not disable the scroll arrow buttons when the region or area is no longer scrollable in this direction. However, it does provide support for you to disable the scroll arrow button under the appropriate conditions.

When scroll arrow buttons are pressed and held, they exhibit a special auto-repeat behavior. This action causes the window to continue scrolling in the associated direction as long as the pointer remains over the arrow button. If the pointer is moved off the arrow button while the user presses the mouse button, the auto-repeat behavior stops and does not continue unless the pointer is moved back over the arrow button (also when the pen tip is moved off the control).

## Scroll Box

The scroll box, sometimes referred to as the elevator, thumb, or slider, moves along the scroll bar to indicate how far the visible portion is from the top (for vertical scroll bars) or from the left edge (for horizontal scroll bars). For example, if the current view is in the middle of a document, the scroll box in the vertical scroll bar is displayed in the middle of the scroll bar.

The size of the scroll box can vary to reflect the difference between what is visible in the window and the entire content of the file, as shown in Figure 6.8.

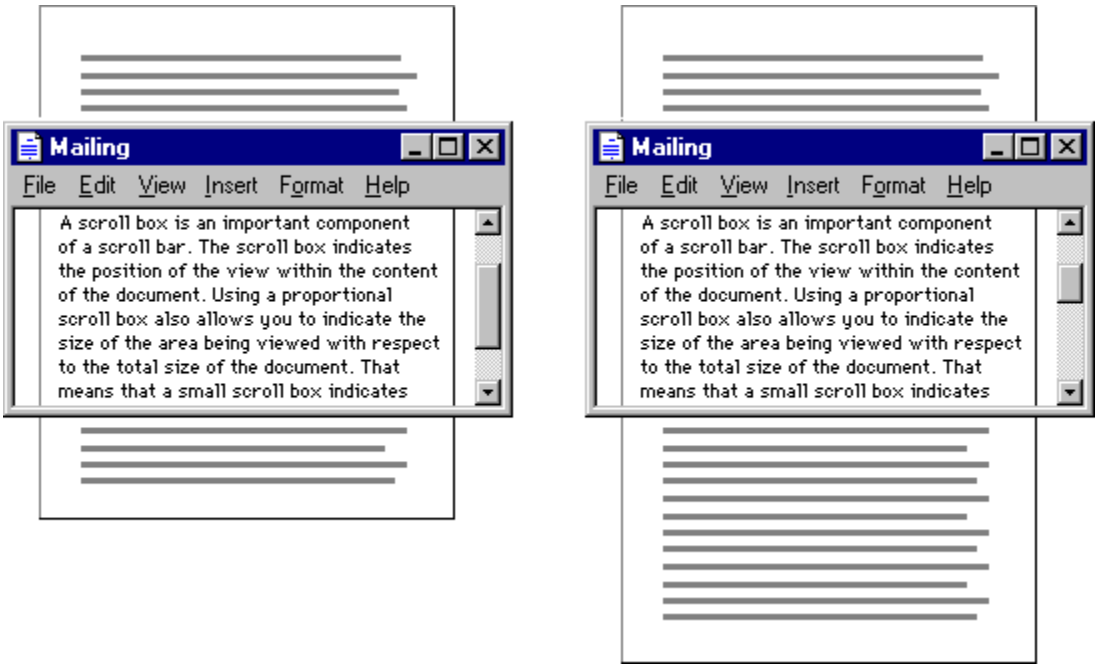



**Figure 6.8 Proportional relationship between scroll box and content**

For example, if the content of the entire document is visible in a window, the scroll box extends the entire length of the scroll bar, and the scroll arrows are disabled. Make the minimum size of the scroll box no smaller than the width of a window's sizing border.

The user can also scroll a window by dragging the scroll box. Update the view continuously as the user moves the scroll box. If you cannot support scrolling at a reasonable speed, you can scroll the information at the end of the drag operation as an alternative.

If the user starts dragging the scroll box and then moves the pointer outside of the scroll bar, the scroll box returns to its original position. The distance the user can move the pointer off the scroll bar before the scroll box snaps back to its original position is proportional to the width of the scroll bar. If dragging ends at this point, the scroll action is canceled — that is, no scrolling occurs. However, if the user moves the pointer back within the scroll-sensitive area, the scroll box returns to tracking the pointer movement. This behavior allows the user to scroll without having to remain within the scroll bar and to selectively cancel the initiation of a drag-scroll operation.

Dragging the scroll box to the end of the scroll bar implies scrolling to the end of that dimension; this does not always mean that the area cannot be scrolled further. If your application's document structure extends beyond the data itself, you can interpret dragging the scroll box to the end of its scroll bar as moving to the end of the data rather than the end of the structure. For example, a typical spreadsheet exceeds the data in it — that is, the spreadsheet may have 65,000 rows, with data only in the first 50 rows. This means you can implement the scroll bar so that dragging the scroll box to the bottom of the vertical scroll bar scrolls to the last row containing data rather than the last row of the spreadsheet. The

user can use the scroll arrow buttons to scroll further to the end of the structure. This situation also illustrates why disabling the scroll arrow buttons can provide important feedback so that the user can distinguish between scrolling to the end of data from scrolling to the end of the extent or structure. In the example of the spreadsheet, when the user drags the scroll box to the end of the scroll bar, the arrow would still be shown as enabled because the user can still scroll further, but it would be disabled when the user scrolls to the end of the spreadsheet.

## Scroll Bar Shaft

The scroll bar shaft not only provides a visual context for the scroll box, it also serves as part of the scrolling interface. Clicking in the scroll bar shaft should scroll the view an equivalent size of the visible area in the direction of the click. For example, if the user clicks in the shaft below the scroll box in a vertical scroll bar, scroll the view a distance equivalent to the height of the view. Where possible, allow overlap from the previous view, as shown in Figure 6.9. For example, if the user clicks below the scroll box, the bottom line becomes the top line of scrolled view. The same thing applies for clicking above the scroll box and horizontal scrolling. These conventions provide the user with a common reference point.

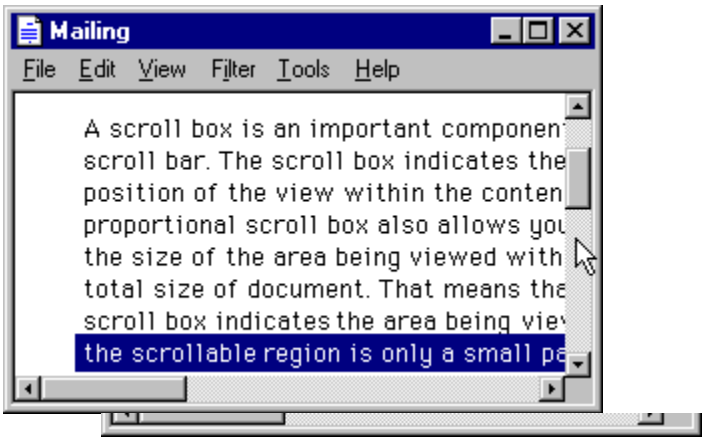



**Figure 6.9 Scrolling with the scroll bar shaft by a screenful**

Pressing and holding mouse button 1 with the pointer in the shaft auto-repeats the scrolling action. If the user moves the pointer outside the scroll-sensitive area while pressing the button, the scrolling action stops. The user can resume scrolling by moving the pointer back into the scroll bar area. (This behavior is similar to the effect of dragging the scroll box.)

## Automatic Scrolling

The techniques previously summarized describe the explicit ways for scrolling. However, the user can also scroll as a secondary result of another user action. This type of scrolling is called *automatic scrolling*. The situations in which to support automatic scrolling are as follows:

- When the user begins or adjusts a selection and drags it past the edge of the scroll bar or window, scroll the area in the direction of the drag.
- When the user drags an object and approaches the edge of a scrollable area, scroll the area following the recommended auto-scroll conventions covered in General Interaction Techniques. Base the scrolling increment on the context of the destination and, if appropriate, on the size of the object being dragged.
- When the user enters text from the keyboard at the edge of a window or moves or copies an object into a location at the edge of a window, the view should scroll to allow the user to focus on the inserted information. The amount to scroll depends on context. For example, for text, vertically scroll a single line at a time. When scrolling horizontally, scroll in units greater than a single character to prevent continuous or uneven scrolling. Similarly, when the user transfers a graphic object near the edge of the view, base scrolling on the size of the object.
- If an operation results in a selection or moves the cursor, scroll the view to display the new selection. For example, for a Find command that selects a matching object, scroll the object into view because usually the user wants to focus on that location. In addition, other forms of navigation may cause scrolling. For example, completing an entry field in a form may result in navigating to the next field. In this case, if the field is not visible, the form can scroll to display it.

## Keyboard Scrolling

Use navigation keys to support scrolling with the keyboard. When the user presses a navigation key, the cursor moves to the appropriate location. For example, in addition to moving the cursor, pressing arrow keys at the edge of a scrollable area scrolls in the corresponding direction. Similarly, the PAGE UP and PAGE DOWN keys are comparable to clicking in the scroll bar shaft, but they also move the cursor.

Optionally, you can use the SCROLL LOCK key to facilitate keyboard scrolling. In this case, when the SCROLL LOCK key is toggled on and the user presses a navigation key, scroll the view without affecting the cursor or selection.

## Placing Adjacent Controls

It is sometimes convenient to locate controls or status bars adjacent to a scroll bar and position the end of the scroll bar to accommodate them. Take care when placing adjacent elements; too many can make it difficult for users to scroll, particularly if you reduce the scroll bar too much. If you need a large number of controls, consider using a conventional toolbar instead.

For more information about toolbars, see Menus, Controls, and Toolbars.

## Splitting Windows

A window can be split into two or more separate viewing areas, which are called *panes*. For example, a split window allows the user to examine two parts of a document at the same time. You can also use a split window to display different, yet simultaneous, views of the same information, as shown in Figure 6.10.

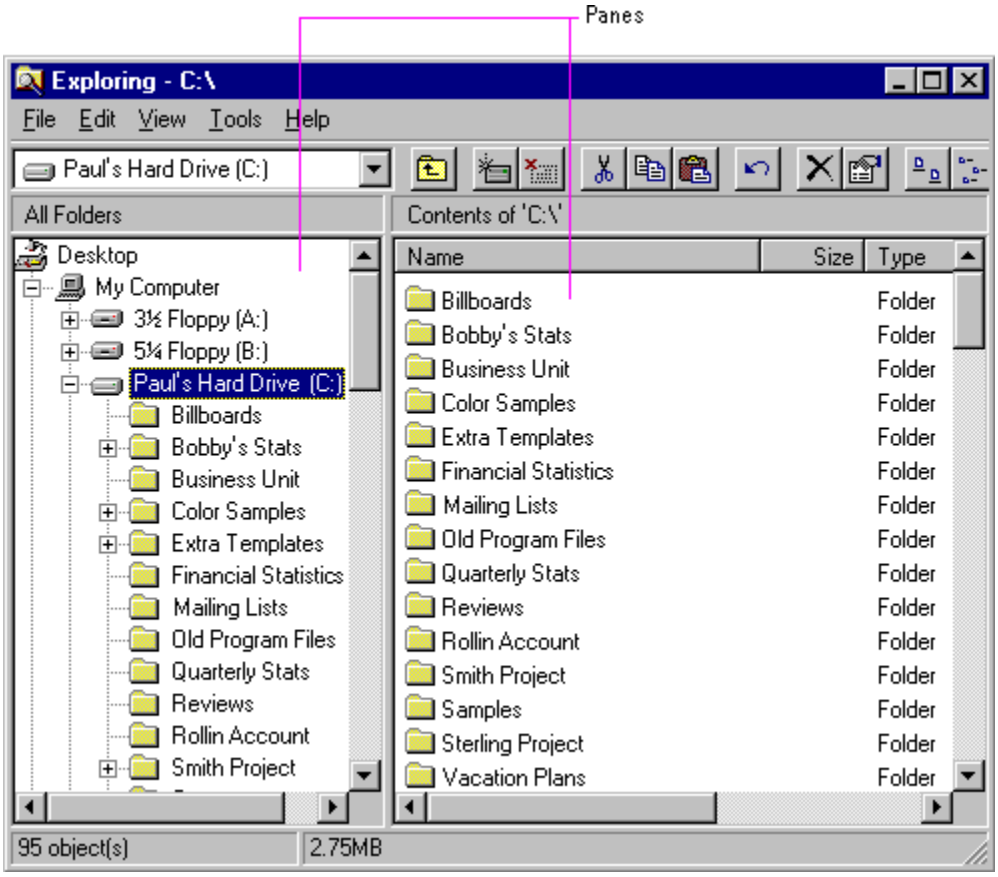



**Figure 6.10 A split window**

While you can use a split window pane to view the contents of multiple files or containers at the same time, displaying these in separate windows typically allows the user to better identify the files as individual elements. When you need to present views of multiple files as a single task, consider window management techniques, such as MDI.

The panes that appear in a split window can be implemented either as part of a window's basic design or as a user-configurable option. To support splitting a window that is not presplit by design, include a split box. A *split box* is a special control placed adjacent to the end of a scroll bar that splits or adjusts the split of a window. The size of the split box should be just large enough for the user to successfully target it with the pointer; the default size of a size handle, such as the window's sizing border, is a good guideline. Locate the split box at the top of the up arrow button of the vertical scroll bar, as shown in Figure 6.11, or to the left of the left arrow button of a horizontal scroll bar.

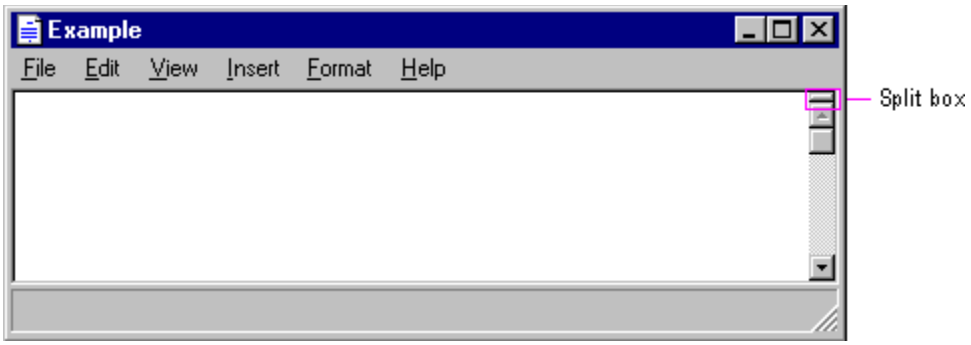



**Figure 6.11 Split box location**

The user splits a window by dragging the split box to the desired position. When the user positions the hot spot of the pointer over a split box, change the pointer's image to provide feedback and help the user target the split box. While the user drags the split box, move a representation of the split box and split bar with the pointer, as shown in Figure 6.12.

At the end of the drag, display a visual separator, called the *split bar*, that extends from one side of the window to the other, defining the edge between the resulting panes, as shown in Figure 6.12. Base the size for the split bar to be, at a minimum, the current setting for the size of window sizing borders. This allows you to appropriately adjust when a user adjusts size borders. If you display the split box after the split operation, place it adjacent to the split bar.

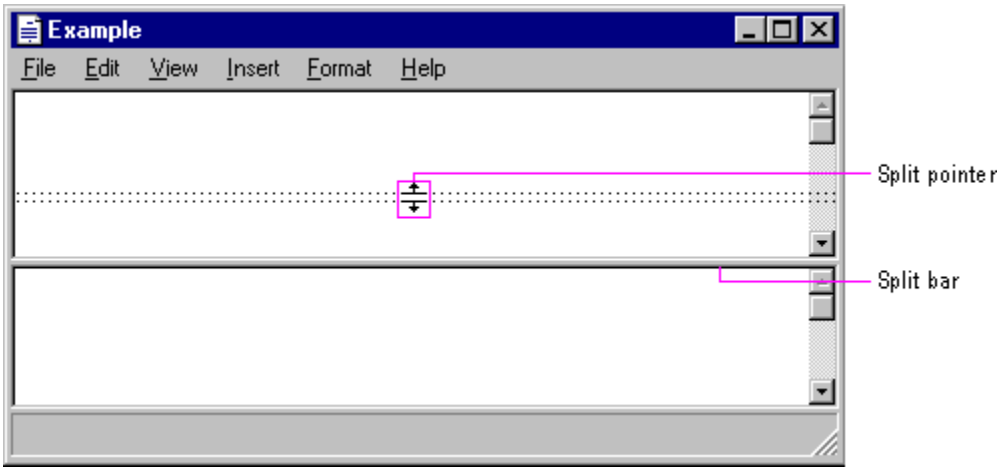



**Figure 6.12 Moving the split bar**

You can support dragging the split bar (or split box) to the end of the scroll bar to close the split. Optionally, you can also support double-clicking (or, for pens, double-tapping) as a shortcut technique for splitting the window at some default location (for example, in the middle of the window or at the last split location) or for removing the split. This technique works best when the resulting window panes display peer views. It may not be appropriate when the design of the window requires that it always be displayed as split or for some types of specialized views.

To provide a keyboard interface for splitting the window, include a Split command on the window or view's menu. When the user chooses the Split command, split the window in the middle or in a context-defined location. Support arrow keys for moving the split box up or down; pressing the ENTER key sets the split at the current location. Pressing the ESC key cancels the split mode.

You can also use other commands to create a split window. For example, you can define specialized views that, when selected by the user, split a window to a fixed or variable set of panes. Similarly, you can enable the user to remove the split of a window by closing a view pane or by selecting another view command.

When the user splits a window, add scroll bars if the resulting panes require scrolling. In addition, you may need to scroll the information in panes so that the split bar does not obscure the content over which it appears. Use a single scroll bar, at the appropriate side of the window, for a set of panes that scroll together. However, if the panes each require independent scrolling, a scroll bar should appear in each pane for that purpose. For example, the vertical scroll bars of a set of panes in a horizontally split window are typically controlled separately.

When you use split window panes to provide separate views, independently maintain each pane's view properties, such as view type and selection state. Display only the selection in the active pane. However, if the selection state is shared across the panes, display a selection in all panes and support selection adjustment across panes.

When a window is closed, save the window's split state (that is, the number of splits, the place where they appear, the scrolled position in each split, and its selection state) as part of the view state information for that window so that it can be restored the next time the window is opened.

# Introduction

Microsoft Windows provides a number of interactive components that make it easy to provide interfaces to carry out commands and specify values. These components also provide a consistent structure and set of interface conventions. This topic describes the interactive elements of menus, controls, and toolbars, and how to use them.

## Menus

*Menus* list commands available to the user. By making commands visible, menus leverage user recognition rather than depending on user recollection of command names and syntax.

There are several types of menus, including drop-down menus, pop-up menus, and cascading menus. The following sections cover these menus in more detail.

## The Menu Bar and Drop-down Menus

A *menu bar*, one of the most common forms of a menu interface, is a special area displayed across the top of a window directly below the title bar (as shown in Figure 7.1). A menu bar includes a set of entries called *menu titles*. Each menu title provides access to a *drop-down menu* composed of a collection of *menu items* or choices.

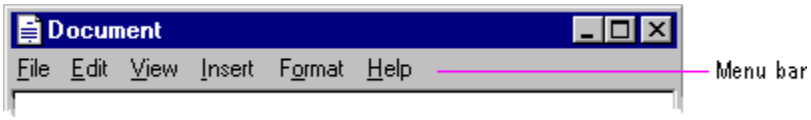



**Figure 7.1 A menu bar**

The content of the menu bar and its drop-down menus are determined by the functionality of your application and the context of a user's interaction. You can also optionally provide user configuration of the menu structure, including hiding the menu bar. If you provide this kind of option, supplement the interface with other components such as pop-up menus, handles, and toolbars, so that a user can access the functionality typically provided by the menu bar.

When displayed, a drop-down menu appears as a panel with its menu items arranged in a column. While the system supports multiple columns for a drop-down menu, avoid this form of presentation because it adds complexity to browsing and interaction of the menu.

## Drop-down Menu Interaction

When the user chooses a menu title, it displays its associated drop-down menu. To display a drop-down menu with the mouse, the user points to the menu title and presses or clicks mouse button 1. This action highlights the menu title and opens the menu. Tapping the menu title with a pen has the same effect as clicking the mouse.

If the user opens a menu by pressing the mouse button while the pointer is over the menu title, the user can drag the pointer over menu items in the drop-down menu. As the user drags, each menu item is highlighted, tracking the pointer as it moves through the menu. Releasing the mouse button with the pointer over a menu item chooses the command associated with that menu item and the system removes the drop-down menu. If the user moves the pointer off the menu and then releases the mouse button, the menu is “canceled” and the drop-down menu is removed. However, if the user moves the pointer back onto the menu before releasing the mouse button, the tracking resumes and the user can still select a menu item.

If the user opens a menu by clicking on the menu title, the menu title is highlighted and the drop-down menu remains displayed until the user clicks the mouse again. Clicking a menu item in the drop-down menu or dragging over and releasing the mouse button on a menu item chooses the command associated with the menu item and removes the drop-down menu. When the system displays a drop-down menu, clicking its associated menu title again cancels the menu and removes the drop-down. Clicking another menu title also results in canceling any displayed drop-down menu, and displays the menu associated with that menu title.

The keyboard interface for drop-down menus uses the ALT key to activate the menu bar. When the user presses an alphanumeric key while holding the ALT key, or after the ALT key is released, the system displays the drop-down menu whose access key for the menu title matches the alphanumeric key (matching is not case sensitive). Pressing a subsequent alphanumeric key chooses the menu item in the drop-down menu with the matching access character.

The user can also use arrow keys to access drop-down menus from the keyboard. When the user presses the ALT key, but has not yet selected a drop-down menu, LEFT ARROW and RIGHT ARROW keys highlight the previous or next menu title, respectively. At the end of the menu bar, pressing another arrow key in the corresponding direction wraps the highlight around to the other end of the menu bar. Pressing the ENTER key displays the drop-down menu associated with the selected menu title. If a drop-down menu is already displayed on that menu bar, then pressing LEFT ARROW or RIGHT ARROW navigates the highlight to the next drop-down menu in that direction, unless the drop-down menu displays its content in multiple columns, in which case the arrow keys move the highlight to the next column in that direction, and then to the next drop-down menu.

Pressing UP ARROW or DOWN ARROW in the menu bar also displays a drop-down menu if none is currently open. In an open drop-down menu, pressing these keys moves to the next menu item in that direction, wrapping the highlight around at the top or bottom. If the drop-down menu has multiple columns, then pressing the arrow keys first wraps the highlight around to the next column.

The user can cancel a drop-down menu by pressing the ALT key whenever the menu bar is active. This not only closes the drop-down menu, it also deactivates the menu bar. Pressing the ESC key also cancels a drop-down menu. However, the ESC key cancels only the current menu level. For example, if a drop-down menu is open, pressing ESC closes the drop-down menu, but leaves its menu title highlighted. Pressing ESC a second time unhighlights the menu title and deactivates the menu bar, returning input focus to the content information in the window.

You can assign shortcut keys to commands in drop-down menus. When the user presses a shortcut key associated with a command in the menu, the command is carried out immediately. Optionally, you can also highlight its menu title, but do not display the drop-down.

## Common Drop-down Menus

This section describes the conventions for drop-down menus commonly used in applications. While these menus are not required for all applications, apply these guidelines when including these menus in your software's interface.

# The File Menu

The File menu provides an interface for the primary operations that apply to a file. Your application should include commands such as Open, Save, Send To, and Print. These commands are often also included on the pop-up menu of the icon displayed in the title bar of the window.

For more information about the commands in the pop-up menu for a title bar icon, see Icon Pop-up Menus.

If your application supports an Exit command, place this command at the bottom of the File menu preceded by a menu separator. When the user chooses the Exit command, close any open windows and files, and stop any further processing. If the object remains active even when its window is closed — for example, like a folder or printer — then include the Close command instead of Exit.

## The Edit Menu

Include general purpose editing commands on the Edit menu. These commands include the Cut, Copy, and Paste transfer commands, OLE object commands, and the following commands (if they are supported).

For more information about transfer commands, see General Interaction Techniques.

| Command | Function |
|---|---|
| Undo | Reverses last action. |
| Repeat | Repeats last action. |
| Find and Replace | Searches for and substitutes text. |
| Delete | Removes the current selection. |
| Duplicate | Creates a copy of the current selection. |

Include these commands on this menu and on the pop-up menu of the selected object.

## The View Menu

Put commands on the View menu that change the user's view of data in the window. Include commands on this menu that affect the view and not the data itself — for example, Zoom or Outline. Also include commands for controlling the display of particular interface elements in the view — for example, Show Ruler. Also place these commands on the pop-up menu of the window or pane.

# The Window Menu

Use the Window menu in multiple document interface-style (MDI) applications for commands associated with managing the windows within an MDI workspace. Also include these commands on the pop-up menu of the parent MDI window.

For more information about the design of MDI software, see Window Management.

## The Help Menu

Use the Help menu for commands that provide access to Help information. Include a Help Topics command; this command provides access to the Help Topics browser, which displays topics included in your application's Help file. Alternatively, you can provide individual commands that access specific pages of the Help Topics browser, such as Contents, Index, and Find Topic. You can also include other user assistance commands on this drop-down menu.

For more information about the Help Topics browser and support for user assistance, see User Assistance.

If you provide access to copyright and version information for your application, include an About *application name* command on this menu. When the user chooses this command, display a window containing the application's name, version number, copyright information, and any other informational properties related to the application. Display this information in a dialog box or alternatively as a copyright page of the property sheet of the application's main executable file. Do not use an ellipsis at the end of this command, because the resulting window does not require the user to provide any further parameters.

## Pop-up Menus

Even if you include a menu bar in your software's interface, you should also support pop-up menus. Pop-up menus provide an efficient way for the user to access the operations of objects, as shown in Figure 7.2. Because pop-up menus are displayed at the pointer's current location, they eliminate the need for the user to move the pointer to the menu bar or a toolbar. In addition, because you populate pop-up menus with commands specific to the object or its immediate context, they reduce the number of commands the user must browse through. Pop-up menus also minimize screen clutter because they are displayed only upon demand and do not require dedicated screen space.

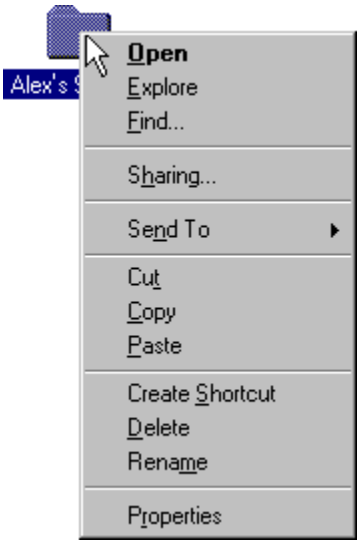



**Figure 7.2 A pop-up menu**

While a pop-up menu looks similar to a drop-down menu, a pop-up menu should only contain commands that apply to the selected object or objects and its context, rather than commands grouped by function. For example, a pop-up menu for a text selection can include commands for moving and copying the text and access to the font properties of the text and the paragraph properties of which the selection is a part. However, keep the size of the pop-up menu as small as possible by limiting the items on the menu to common, frequent actions. It is better to include a single Properties command and allow the user to navigate among properties in the resulting property sheet than to list individual properties in the pop-up menu.

The container or the composition of which a selection is a part typically supplies the pop-up menu for the selection. Similarly, the commands included on a pop-up menu may not always be supplied by the object itself, but rather be a combination of those commands provided by the object and by its current container. For example, the pop-up menu for a file in a folder includes transfer commands. In this case, the folder (container) supplies the commands, not the files. Pop-up menus for OLE objects follow these same conventions.

Avoid using a pop-up menu as the exclusive means to a particular operation. At the same time, the items in a pop-up menu need not be limited only to commands that are provided in drop-down menus.

When ordering the commands in a pop-up menu, use the following guidelines:

- Place the object's primary commands first (for example, commands such as Open, Play, and Print), transfer commands, other commands supported by the object (whether provided by the object or by its context), and the What's This? command (when supported).
- Order the transfer commands as Cut, Copy, Paste, and other specialized Paste commands.
- Place the Properties command, when present, as the last command on the menu.

For more information about transfer commands and the Properties command, see General Interaction Techniques. For more information about the What's This? command, see User Assistance.

## Pop-up Menu Interaction

With a mouse, the user displays a pop-up menu by clicking an object with button 2. The down transition of the mouse button selects the object. Upon the up transition, display the menu to the right and below the hot spot of the pointer adjusted to avoid the menu being clipped by the edge of the screen.

If the pointer is over an existing selection when the user invokes a pop-up menu, display the menu that applies to that selection. If the menu is outside a selection but within the same selection scope, then establish a new selection (usually resetting the current selection in that scope) at the button down point and display the menu for the new selection. Dismiss the pop-up menu when the user clicks outside the menu with button 1 or if the user presses the ESC key.

You can support pop-up menus for objects that are implicitly selected or cannot be directly selected, such as scroll bars or items in a status bar. When providing pop-up menus for objects such as controls, include commands for the object that the control represents, rather than for the control itself. For example, a scroll bar represents a navigational view of a document, so commands might include Beginning of Document, End of Document, Next Page, and Previous Page. But when a control represents itself as an object, as in a forms layout or window design environment, you can include commands that apply to the control—for example, commands to move or copy the control.

The pen interface uses an action handle in pen-enabled controls to access the pop-up menu for the selection. Tapping the action handle displays the pop-up menu, as shown in Figure 7.3.

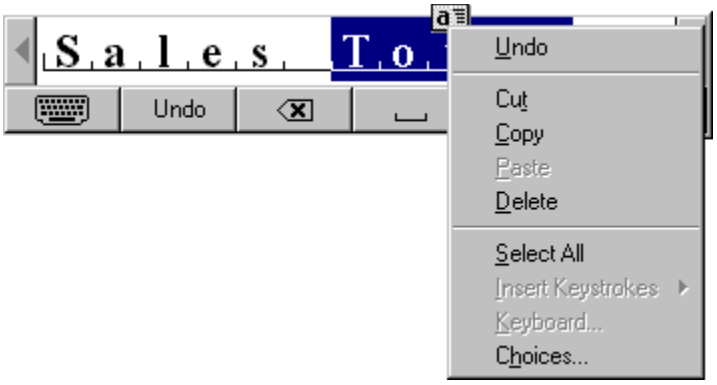



**Figure 7.3 Using an action handle to provide pen access to pop-up menus**

In addition, you can use techniques like barrel-tapping or the pop-up menu gesture to display a pop-up menu. This interaction is equivalent to a mouse button 2 click.

Use SHIFT+F10 and the Application key (for keyboards that support the Windows keys specification) to provide keyboard access for pop-up menus. In addition, menu access keys, arrow keys, ENTER, and ESC keys all operate in the same fashion in the menu as they do in drop-down menus. To enhance space and visual efficiency, avoid including shortcut keys in pop-up menus.

The system provides a message, WM_CONTEXTMENU, when the user presses a system defined pop-up menu key. For more information about this message, see documentation included in the Microsoft Win32 Software Development Kit (SDK).

## Common Pop-up Menus

The pop-up menus included in any application depend on the objects and context supplied by that application. The following sections describe common pop-up menus for Windows-based applications.

## The Window Pop-up Menu

The window pop-up menu is the pop-up menu associated with a window — do not confuse it with the Window drop-down menu found in MDI applications. The window pop-up menu replaces the Windows 3.1 Control menu, also referred to as the System menu. For example, a typical primary window includes Close, Restore, Move, Size, Minimize, and Maximize.

You can also include other commands on the window's menu that apply to the window or the view within the window. For example, an application can append a Split command to the menu to facilitate splitting the window into panes. Similarly, you can add commands that affect the view, such as Outline, commands that add, remove, or filter elements from the view, such as Show Ruler, or commands that open certain subordinate or special views in secondary windows, such as Show Color Palette.

A secondary window also includes a pop-up menu. Usually, because the range of operations are more limited than in a primary window, a secondary window's pop-up menu includes only Move and Close commands, or just Move. Palette windows can also include an Always on Top command that sets the window to always be on top of its parent window and secondary windows of its parent window.

The user displays a window's pop-up menu by clicking mouse button 2 anywhere in the title bar area, excluding the title bar icon. Clicking on the title bar icon with button 2 displays the pop-up menu for the object represented by the icon. To avoid confusing users, if you do not provide a pop-up menu for the title bar icon, do not display the pop-up for the window when the user clicks with button 2 on the title bar icon.

For compatibility with previous versions of Windows, the system also supports clicking button 1 on the icon in the title bar to access the pop-up menu of a window. However, do not document this as the primary method for accessing the pop-up menu for the window. Document only the button 2 technique.

For the pen, performing barrel-tapping or the equivalent pop-up menu gesture on these areas displays the menu. Pressing ALT+SPACEBAR also displays the menu. The pop-up for the window can also be accessed from the keyboard by the user pressing the ALT key and then using the arrow keys to navigate beyond the first or last entry in the menu bar. In MDI applications, the pop-up menu for a child window can also be accessed this way or directly using ALT+HYPHEN.

## Icon Pop-up Menus

Pop-up menus displayed for icons include operations of the objects represented by those icons. Accessing the pop-up menu of an application or document icon follows the standard conventions for pop-up menus, such as displaying the menus with a mouse button 2 click.

An icon's container application supplies the pop-up menu for the icon. For example, pop-up menus for icons placed in standard folders or on the desktop are automatically provided by the system. However, your application supplies the pop-up menus for OLE embedded or linked objects placed in it — that is, placed in the document or data files your application supports.

For more information about supporting pop-up menus for OLE objects, see Working with OLE Embedded and OLE Linked Objects.

The container populates the pop-up menu for an icon with commands the container supplies for its content, such as transfer commands and those registered by the object's type. For example, an application can register a New command that automatically generates a new data file of the type supported by the application.

For more information about registering commands, see Integrating with the System.

The pop-up menu of an application's icon, for example, the Microsoft WordPad executable file, should include the commands listed in Table 7.1.

**Table 7.1 Application File Icon Pop-up Menu Commands**

| Command | Meaning |
|---|---|
| Open | Opens the application file. |
| Send To | Displays a submenu of destinations to which the file can be transferred. The content of the submenu is based on the content of the system's Send To folder. |
| Cut | Marks the file for moving. (Registers the file on the Clipboard.) |
| Copy | Marks the file for duplication. (Registers the file on the Clipboard.) |
| Paste | Attempts to open the file registered on the Clipboard with the application. |
| Create Shortcut | Creates a shortcut icon of the file. |
| Delete | Deletes the file. |
| Rename | Allows the user to edit the filename. |
| Properties | Displays the properties for the file. |

An icon representing a document or data file typically includes the following common menu items for the pop-up menu for its icon.

**Table 7.2 Document or Data File Icon Pop-up Menu Commands**

| Command | Meaning |
|---|---|
| Open | Opens the file's primary window. |
| Print | Prints the file on the current default printer. |
| Quick View | Displays the file using a special viewing tool window. |
| Send To | Displays a submenu of destinations to which the file can be transferred. The content of the submenu is based on |

| | |
|---|---|
| | the content of the system's Send To folder. |
| Cut | Marks the file for moving. (Registers the file on the Clipboard.) |
| Copy | Marks the file for duplication. (Registers the file on the Clipboard.) |
| Delete | Deletes the file. |
| Rename | Allows the user to edit the filename. |
| Properties | Displays the properties for the file. |

With the exception of the Open and Print commands, the system automatically provides these commands for icons when they appear in system containers, such as the desktop or folders. If your application supplies its own containers for files, you need to supply these commands.

For the Open and Print commands to appear on the menu, your application must register these commands in the system registry. You can also register additional or replacement commands. For example, you can optionally register a Quick View command that displays the content of the file without running the application and a What's This? command that displays descriptive information for your data file types.

For more information about registering commands and the Quick View command, see Integrating with the System. For more information about the What's This? command, see User Assistance.

The icon in the title bar of a window represents the same object as the icon the user opens. As a result, the application associated with the icon also includes a pop-up menu with appropriate commands for the title bar's icon. When the icon of an application appears in the title bar, include the same commands on its pop-up menu as are included for the icon that the user opens, unless a particular command cannot be applied when the application's window is open. In addition, replace the Open command with Close.

Similarly, when the icon of the data or document file appears in the title bar, you also use the same commands as found on its file icon, with the following exceptions: replace the Open command with a Close command and add Save if the edits in the document require explicit saving to file.

For an MDI application, supply a pop-up menu for the application icon in the parent window, following the conventions for application title bar icons. Also consider including the following commands where they apply.

For more information about the design of MDI-style applications, see Window Management.

**Table 7.3 Optional MDI Parent Window Title Bar Icon Pop-up Menu Commands**

| Command | Meaning |
|---|---|
| New | Creates a new data file or displays a list of data file types supported by the application from which the user can choose. |
| Save All | Saves all data files open in the MDI workspace, and the state of the MDI window. |
| Find | Displays a window that allows the user to specify criteria to locate a data file. |

In addition, supply an appropriate pop-up menu for the title bar icon that appears in the child window's title bar. You can follow the same conventions for non-MDI data files.

## Cascading Menus

A *cascading menu* (also referred to as a *hierarchical menu* or child menu) is a submenu of a menu item. The visual cue for a cascading menu is the inclusion of a triangular arrow display adjacent to the label of its parent menu item.

You can use cascading menus to provide user access to additional choices rather than taking up additional space in the parent menu. They may also be useful for displaying hierarchically related objects.

Be aware that cascading menus can add complexity to the menu interface by requiring the user to navigate further through the menu structure to get to a particular choice. Cascading menus also require more coordination to handle the changes in direction necessary to navigate through them.

In light of these design tradeoffs, use cascading menus sparingly. Minimize the number of levels for any given menu item, ideally limiting your design to a single submenu. Avoid using cascading menus for frequent, repetitive commands.

As an alternative, make choices available in a secondary window, particularly when the choices are independent settings; this allows the user to set multiple options in one invocation of a command. You can also support many common options as entries on a toolbar.

The user interaction for a cascading menu is similar to that of a drop-down menu from the menu bar, except a cascading menu displays after a short time-out. This avoids the unnecessary display of the menu if the user is browsing or navigating to another item in the parent menu. Once displayed, if the user moves the pointer to another menu item, the cascading menu is removed after a short time-out. This time-out enables the user to directly drag from the parent menu into an entry in its cascading menu.

## Menu Titles

All drop-down and cascading menus have a menu title. For drop-down menus, the menu title is the entry that appears in the menu bar. For cascading menus, the menu title is the name of the parent menu item. Menu titles represent the entire menu and should communicate as clearly as possible the purpose of all items on the menu.

Use single words for menu bar menu titles. Multiple word titles or titles with spaces may be indistinguishable from two one-word titles. In addition, avoid uncommon compound words, such as Fontsize.

Define one character of each menu title as its access key. This character provides keyboard access to the menu. Windows displays the access key for a menu title as an underlined character, as shown in Figure 7.4.

For more information about keyboard input and defining access keys, see Input Basics.

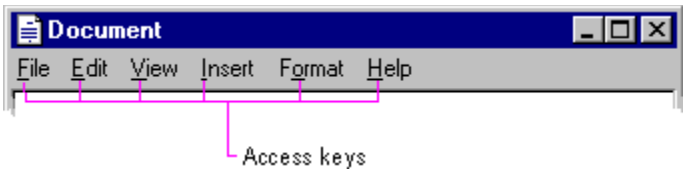



**Figure 7.4 Access keys in a menu bar**

Define unique access keys for each menu title. Using the same access key for more than one menu title may eliminate direct access to a menu.

## Menu Items

Menu items are the individual choices that appear in a menu. Menu items can be text, graphics — such as icons — or graphics and text combinations that represent the actions presented in the menu. The format for a menu item provides the user with visual cues about the nature of the effect it represents, as shown in Figure 7.5.

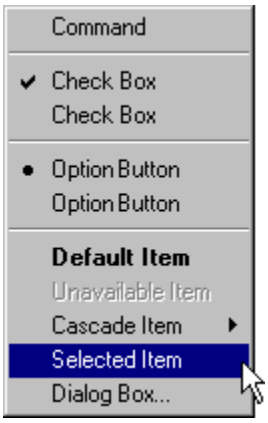



**Figure 7.5 Formats for different menu items**

Whenever a menu contains a set of related menu items, you can separate those sets with a grouping line known as a *separator*. The standard separator is a single line that spans the width of the menu. Avoid using menu items themselves as group separators, as shown in Figure 7.6.

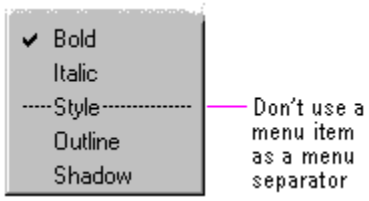



**Figure 7.6 Inappropriate separator**

Always provide the user with a visual indication about which menu items can be applied. If a menu item is not appropriate or applicable in a particular context, then disable or remove it. Leaving the menu item enabled and presenting a message box when the user selects the menu item is a poor method for providing feedback.

In general, it is better to disable a menu item rather than remove it because this provides more stability in the interface. However, if the context is such that the menu item is no longer or never relevant, remove it. For example, if a menu displays a set of open files and one of those files is closed or deleted, it is appropriate to remove the corresponding menu item.

If all items in a menu are disabled, disable its menu title. If you disable a menu item or its title, it does not prevent the user from browsing or choosing it. If you provide status bar messages, you can display a message indicating that the command is unavailable and why.

The system provides a standard appearance for displaying disabled menu items. If you are supplying your own visuals for a disabled menu item, follow the visual design guidelines for how to display it with an unavailable appearance.

For more information about displaying commands with an unavailable appearance, see Visual Design.

## Types of Menu Items

Many menu items take effect as soon as they are chosen. If the menu item is a command that requires additional information to complete its execution, follow the command with an *ellipsis* (...). The ellipsis informs the user that information is incomplete. When used with a command, it indicates that the user needs to provide more information to complete that command. Such commands usually result in the display of a dialog box. For example, the Save As command includes an ellipsis because the command is not complete until the user supplies or confirms a filename.

Not every command that produces a dialog box or other secondary window should include an ellipsis. For example, do not include an ellipsis with the Properties command because carrying out the Properties command displays a properties window. After completing the command, no further parameters or actions are required to fulfill the intent of the command. Similarly, do not include an ellipsis for a command that may result in the display of a message box.

While you can use menu items to carry out commands, you can also use menu items to switch a mode or set a state or property, rather than initiating a process. For example, choosing an item from a menu that contains a list of tools or views implies changing to that state. If the menu item represents a property value, when the user chooses the menu item, the property setting changes.

Menu items for state settings can be independent or interdependent:

- Independent settings are the menu equivalent of check boxes. For example, if a menu contains text properties, such as Bold and Italic, they form a group of independent settings. The user can change each setting without affecting the others, even though they both apply to a single text selection. Include a check mark to the left of an independent setting when that state applies.
- Interdependent settings are the menu equivalent of option buttons. For example, if a menu contains alignment properties — such as Left, Center, and Right — they form a group of interdependent settings. Because a particular paragraph can have only one type of alignment, choosing one resets the property to be the chosen menu item setting. When the user chooses an interdependent setting, place an option button mark to the left of that menu item.

When using the menu to represent the two states of a setting, if those states are obvious opposites, such as the presence or absence of a property value, you can use a check mark to indicate when the setting applies. For example, when reflecting the state of a text selection with a menu item labeled Bold, show a check mark next to the menu item when the text selection is bold and no check mark when it is not. If a selection contains mixed values for the same stat reflected in the menu, you also display the menu without the check mark.

However, if the two states of the setting are not obvious opposites, use a pair of alternating menu item names to indicate the two states. For example, a naive user might guess that the opposite of a menu item called Full Duplex is Empty Duplex. Because of this ambiguity, pair the command with the alternative name Half Duplex, rather using a mark to indicate the alternative states, and consider the following guidelines for how to display those alternatives:

- If there is room in a menu, include both alternatives as individual menu items and interdependent choices. This avoids confusion because the user can view both options simultaneously. You can also use menu separators to group the choices.

Avoid defining menu items that change depending on the state of a modifier key. Such techniques hide functionality from a majority of users.

- If there is not sufficient room in the menu for the alternative choices, you can use a single menu item and change its name to the alternative action when selected. In this case, the menu item's name does not reflect the current state; it indicates the state after choosing the item. Where possible, define names that use the same access key. For example, the letter D could be used for a menu item that toggles between Full Duplex and Half Duplex.

A menu can also have a default item. A default menu item reflects a choice that is also supported through a shortcut technique, such as double-clicking or drag and drop. For example, if the default command for an icon is Open, define this as the default menu item. Similarly, if the default command

for a drag and drop operation is Copy, display this command as the default menu item in the pop-up menu that results from a nondefault drag and drop operation (button 2). The system designates a default menu item by displaying its label as bold text.

## Menu Item Labels

Include descriptive text or a graphic label for each menu item. Even if you provide a graphic for the label, consider including text as well. The text allows you to provide more direct keyboard access to the user and provides support for a wider range of users.

Use the following guidelines for defining text menu names for menu item labels:

- Define unique item names within a menu. However, item names can be repeated in different menus to represent similar or different actions.
- Use a single word or multiple words, but keep the wording brief and succinct. Verbose menu item names can make it harder for the user to scan the menu.
- Define unique access keys for each menu item within a menu. This provides the user direct keyboard access to the menu item. The guidelines for selecting an access key for menu items are the same as for menu titles, except that the access key for a menu item can also be a number included at the beginning of the menu item name. This is useful for menu items that vary, such as filenames. Where possible, also define consistent access keys for common commands.

For more information about defining access keys, see Input Basics. For more information about common access key assignments, see Keyboard Interface Summary.

- Follow book title capitalization rules for menu item names. For English language versions, capitalize the first letter of every word, except for articles, conjunctions, and prepositions that occur other than at the beginning or end of a multiple-word name. For example, the following menu names are correct: New Folder, Go To, Select All, and Table of Contents.
- Avoid formatting individual menu item names with different text properties. Even though these properties illustrate a particular text style, they also may make the menu cluttered, illegible, or confusing. For example, it may be difficult to indicate an access key if an entire menu entry is underlined.

# Shortcut Keys in Menu Items

If you define a keyboard shortcut associated with a command in a drop-down menu, display the shortcut in the menu. Display the shortcut key next to the item and align shortcuts with other shortcuts in the menu. Left align at the first tab position after the longest item in the menu that has a shortcut. Do not use spaces for alignment because they may not display properly in the proportional font used by the system to display menu text or when the font setting menu text changes.

You can match key names with those commonly inscribed on the keycap. Display CTRL and SHIFT key combinations as Ctrl+*key* (rather than Control+*key* or CONTROL+*key* or ^+*key*) and Shift+*key*. When using function keys for menu item shortcuts, display the name of the key as F*n*, where *n* is the function key number.

For more information about the selection of shortcut keys, see Input Basics.

Avoid including shortcut keys in pop-up menus. Pop-up menus are already a shortcut form of interaction and are typically accessed with the mouse. In addition, excluding shortcut keys makes pop-up menus easier for users to scan.

# Controls

*Controls* are graphic objects that represent the properties or operations of other objects. Some controls display and allow editing of particular values. Other controls start an associated command.

Each control has a unique appearance and operation designed for a specific form of interaction. The system also provides support for designing your own controls. When defining your own controls, follow the conventions consistent with those provided by the system-supplied controls.

For more information about using standard controls and designing your own controls, see Visual Design.

Like most elements of the interface, controls provide feedback indicating when they have the input focus and when they are activated. For example, when the user interacts with controls using a mouse, each control indicates its selection upon the down transition of the mouse button, but does not activate until the user releases the button, unless the control supports auto-repeat.

Controls are generally interactive only when the pointer, actually the hot spot of the pointer, is over the control. If the user moves the pointer off the control while pressing a mouse button, the control no longer responds to the input device. If the user moves the pointer back onto the control, it once again responds to the input device. The hot zone, or boundary that defines whether a control responds to the pointer, depends on the type of control. For some controls, such as buttons, the hot zone coincides with the visible border of the control. For others, the hot zone may include the control's graphic and label (for example, check boxes) or some controls, such as scroll bars, as defined around the control's borders.

Many controls provide labels. Because labels help identify the purpose of a control, always label a control with which you want the user to directly interact. If a control does not have a label, you can provide a label using a static text field or a tooltip control. Define an access key for text labels to provide the user direct keyboard access to a control. Where possible, define consistent access keys for common commands.

For more information about defining access keys, see Input Basics.

While controls provide specific interfaces for user interaction, you can also include pop-up menus for controls. This can provide an effective way to transfer the value the control represents or to provide access to context-sensitive Help information. The interface to pop-up menus for controls follows the standard conventions for pop-up menus, except that it does not affect the state of the control; that is, clicking the control with button 2 does not trigger the action associated with the control when the user clicks it with button 1. The only action is the display of the pop-up menu.

A pop-up menu for a control is contextual to what the control represents, rather than the control itself. Therefore, avoid commands such as Set, Unset, Check, or Uncheck. The exception is in a forms design or window layout context, where the commands on the pop-up menu can apply to the control itself.

## Buttons

Buttons are controls that start actions or change properties. There are three basic types of buttons: command buttons, option buttons, and check boxes.

## Command Buttons

A *command button*, also referred to as a push button, is a control, commonly rectangular in shape, that includes a label (text, graphic, or sometimes both), as shown in Figure 7.7.


**Figure 7.7 Command buttons**

When the user chooses a command button with mouse button 1 (for pens, tapping), the command associated with the button is carried out. When the user presses the mouse button, the input focus moves to the button, and the button state changes to its pressed appearance. If the user moves the pointer off the command button while the mouse button remains pressed, the button returns to its original state. Moving the pointer back over the button while pressing the mouse button returns the button to its pressed state.

When the user releases the mouse button with the pointer on the command button, the command associated with the control starts. If the pointer is not on the control when the user releases the mouse button, no action occurs.

You can define access keys and shortcut keys for command buttons. In addition, you can use the TAB key and arrow keys to support user navigation to or between command buttons. The SPACEBAR activates a command button if the user moves the input focus to the button.

For more information about navigation and activation of controls, see Secondary Windows.

The effect of choosing a button is immediate with respect to its context. For example, in toolbars, clicking a button carries out the associated action. In a secondary window, such as a dialog box, activating a button may initiate a transaction within the window, or apply a transaction and close the window.

The command button's label represents the action the button starts. When using a text label, the text should follow the same capitalization conventions defined for menus. If the control is disabled, display the label of the button as unavailable.

Include an ellipsis (...) as a visual cue for buttons associated with commands that require additional information. Like menu items, the use of an ellipsis indicates that further information is needed, not simply that a window will appear. Some buttons, when clicked, can display a message box, but this does not imply that the command button's label should include an ellipsis.

You can use command buttons to enlarge a secondary window and display additional options, also known as an *unfold button*. An unfold button is not really a different type of control, but the use of a command button for this specific function. When using a command button for this purpose, include a pair of "greater than" (>>) characters as part of the button's label.

In some cases, a command button can represent an object and its default action. For example, the taskbar buttons represent an object's primary window and the Restore command. When the user clicks on the button with mouse button 1, the default command of the object is carried out. Clicking on a button with mouse button 2 displays a pop-up menu for the object the button represents.

You can also use command buttons to reflect a mode or property value similar to the use of option buttons or check boxes. While the typical interaction for a command button is to return to its normal "up" state, if you use it to represent a state, display the button in the option-set appearance, as shown in Table 7.4.

For more information about the appearance of different states of buttons, see Visual Design.

**Table 7.4 Command Button Appearance**

| Appearance | Button state |
|---|---|
|  | Normal appearance |
|  | Pressed appearance |
|  | Option-set appearance |
|  | Unavailable appearance |
|  | Option-set, unavailable appearance |
|  | Mixed-value appearance |
|  | Input focus appearance |

You can also use command buttons to set tool modes — for example, in drawing or forms design programs for drawing out specific shapes or controls. In this case, design the button labels to reflect the tool's use. When the user chooses the tool (that is, clicks the button), display the button using the option-set appearance and change the pointer to indicate the change of the mode of interaction.

You can also use a command button to display a pop-up menu. This convention is known as a *menu button*. While this is not a specific control provided by the system, you can create this interface using the standard components.

A menu button looks just like a standard command button, except that, as a part of its label, it includes a triangular arrow similar to the one found in cascading menu titles, as shown in Figure 7.8.

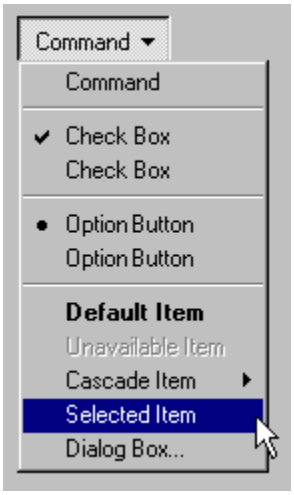



**Figure 7.8 A menu button**

A menu button supports the same type of interaction as a drop-down menu; the menu is displayed when the button is pressed and allows the user to drag into the menu from the button and make menu

selections. Like any other menu, use highlighting to track the movement of the pointer.

Similarly, when the user clicks a menu button, the menu is displayed. At this point, interaction with the menu is the same as with any menu. For example, clicking a menu item carries out the associated command. Clicking outside the menu or on the menu button removes the menu.

When pressed, display the menu button with the pressed appearance. When the user releases the mouse button and the menu is displayed, use the option-set appearance. Otherwise, the menu button's appearance is the same as a typical command button. For example, if the button is disabled, display the button using the unavailable appearance.

## Option Buttons

An *option button*, also referred to as a radio button, represents a single choice within a limited set of mutually exclusive choices — that is, in any group of option buttons, only one option in the group can be set. Accordingly, always group option buttons in sets of two or more, as shown in Figure 7.9.

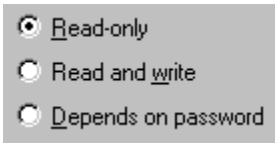



**Figure 7.9 A set of option buttons**

Option buttons appear as a set of small circles. When an option button choice is set, a dot appears in the middle of the circle. When the choice is not the current setting, the circle is empty. Avoid using option buttons to start an action other than the setting of a particular option or value represented by the option button. The only exception is that you can support double-clicking the option button as a shortcut for setting the value and carrying out the default command of the window in which the option buttons appear, if choosing an option button is the primary user action for the window.

You can use option buttons to represent a set of choices for a particular property. When the option buttons reflect a selection with mixed values for that property, display all the buttons in the group using the mixed-value appearance to indicate that multiple values exist for that property. The mixed-value appearance for a group of option buttons displays all buttons without a setting dot, as shown in Figure 7.10.

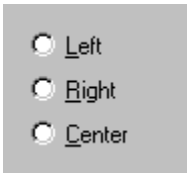



**Figure 7.10 Option buttons with mixed-value appearance**

If the user chooses any option button in a group with mixed-value appearance, that value becomes the setting for the group; the dot appears in that button and all the other buttons in the group remain empty.

Limit the use of option buttons to small sets of options, typically seven or less, but always at least two. If you need more choices, consider using another control, such as a single selection list box or drop-down list box.

Each option button includes a text label. (If you need graphic labels for a group of exclusive choices, consider using command buttons instead.) The standard control allows you to include multiple line labels. When implementing multiple line labels, use top alignment, unless the context requires an alternate orientation.

Define the option button's label to represent the value or effect for that choice. Also use the label to indicate when the choice is unavailable. Use sentence capitalization for an option button's label; only capitalize the first letter of the first word, unless it is a word in the label normally capitalized.

For more information about labeling or appearance states, see Visual Design.

Because option buttons appear as a group, you can use a group box control to visually define the group. You can label the option buttons to be relative to a group box's label. For example, for a group box labeled Alignment, you can label the option buttons as Left, Right, and Center.

As with command buttons, the mouse interface for choosing an option button uses a click with mouse

button 1 (for pens, tapping) either on the button's circle or on the button's label. The input focus is moved to the option button's label when the user presses the mouse button, and the option button displays its pressed appearance. If the user moves the pointer off the option button before releasing the mouse button, the option button is returned to its original state. The option is not set until the user releases the mouse button while the pointer is over the control. Releasing the mouse button outside of the option button or its label has no effect on the current setting of the option button. In addition, successive mouse clicks on the same option button do not toggle the button's state; the user needs to explicitly select an alternative choice in the group to change or restore a former choice.

Assign access keys to option button labels to provide a keyboard interface to the buttons. You can also define the TAB or arrow keys to allow the user to navigate and choose a button. Access keys or arrow keys automatically set an option button and set the input focus to that button.

For more information about the guidelines for defining access keys, see Input Basics. For more information about navigation and interaction with option buttons, see Secondary Windows.

## Check Boxes

Like option buttons, check boxes support options that are either on or off; check boxes differ from option buttons in that you typically use check boxes for independent or nonexclusive choices. As in the case of independent settings in menus, use check boxes only when both states of the choice are clearly opposite and unambiguous. If this is not the case, then use option buttons or some other form of single selection choice control instead.

A check box appears as a square box with an accompanying label. When the choice is set, a check mark appears in the box. When the choice is not set, the check box is empty, as shown in Figure 7.11.

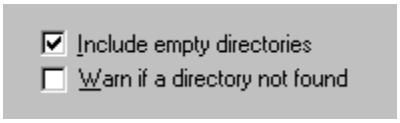



**Figure 7.11 A set of check boxes**

A check box's label is typically displayed as text and the standard control includes a label. (Use a command button instead of a check box when you need a nonexclusive choice with a graphic label.) Use a single line of text for the label as this makes the label easier to read. However, if you do use multiple lines, use top alignment, unless the context requires a different orientation.

Define a check box's label to appropriately express the value or effect of the choice. Use sentence capitalization for multiple word labels. The label also serves as an indication of when the control is unavailable.

Group related check box choices. If you group check boxes, it does not prevent the user from setting the check boxes on or off in any combination. While each check box's setting is typically independent of the others, you can use a check box's setting to affect other controls. For example, you can use the state of a check box to filter the content of a list. If you have a large number of choices or if the number of choices varies, use a multiple selection list box instead of check boxes.

When the user clicks a check box with mouse button 1 (for pens, tapping) either on the check box square or on the check box's label, that button is chosen and its state is toggled. When the user presses the mouse button, the input focus moves to the control and the check box assumes its pressed appearance. Like option buttons and other controls, if the user moves the pointer off the control while holding down the mouse button, the control's appearance returns to its original state. The setting state of the check box does not change until the mouse button is released. To change the control's setting, the pointer must be over the check box or its label when the user releases the mouse button.

Define access keys for check box labels to provide a keyboard interface for navigating to and choosing a check box. In addition, the TAB key and arrow keys can also be supported to provide user navigation to or between check boxes. In a dialog box, for example, the SPACEBAR toggles a check box when the input focus is on the check box.

For more information about guidelines for defining access keys, see Input Basics. For more information about navigation and supporting interaction for controls with the keyboard, see Secondary Windows.

If you use a check box to display the value for the property of a multiple selection whose values for that property differ (for example, for a text selection that is partly bold), display the check box in its mixed-value appearance, as shown in Figure 7.12.


**Figure 7.12 A mixed-value check box (magnified)**

If the user chooses a check box in the mixed-value state, the associated value is set and a check mark is placed in it. This implies that the property of all elements in the multiple selection will be set to this value when it is applied. If the user chooses the check box again, the setting to be unchecked is toggled. If applied to the selection, the value will not be set. If the user chooses the check box a third time, the value is toggled back to the mixed-value state. When the user applies the value, all elements in the selection retain their original value. This three-state toggling occurs only when the control represents a mixed set of values.

# List Boxes

A *list box* is a convenient, preconstructed control for displaying a list of choices for the user. The choices can be text, color, icons, or other graphics. The purpose of a list box is to display a collection of items and, in most cases, support selection of a choice of an item or items in the list.

List boxes are best for displaying large numbers of choices that vary in number or content. If a particular choice is not available, omit the choice from the list. For example, if a point size is not available for the currently selected font, do not display that size in the list.

Order entries in a list using the most appropriate choice to represent the content in the list and to facilitate easy user browsing. For example, alphabetize a list of names, but put a list of dates in chronological order. If there is no natural or logical ordering for the content, use ascending or alphabetical ordering — for example, 0–9 or A–Z.

List box controls do not include their own labels. However, you can include a label using a static text field; the label enables you to provide a descriptive reference for the control and keyboard access to the control. Use sentence capitalization for multiple word labels and make certain that your support for keyboard access moves the input focus to the list box and not the static text field label.

For more information about navigation to controls in a secondary window, see Secondary Windows. For more information about defining access keys for control labels, see Input Basics. For more information about static text fields, see Static Text Fields.

When a list box is disabled, display its label using an unavailable appearance. If possible, display all of the entries in the list as unavailable to avoid confusing the user as to whether the control is enabled or not.

The width of the list box should be sufficient to display the average width of an entry in the list. If that is not practical because of space or the variability of what the list might include, consider one or more of the following options:

- Make the list box wide enough to allow the entries in the list to be sufficiently distinguished.
- Use an ellipsis (...) in the middle or at the end of long text entries to shorten them, while preserving the important characteristics needed to distinguish them. For example, for long paths, usually the beginning and end of the path are the most critical; you can use an ellipsis to shorten the entire name: \Sample\...\Example.
- Include a horizontal scroll bar. However, this option reduces some usability, because adding the scroll bar reduces the number of entries the user can view at one time. In addition, if most entries in the list box do not need to be horizontally scrolled, including a horizontal scroll bar accommodates the infrequent case.

When the user clicks an item in a list box, it becomes selected. Support for multiple selection depends on the type of list box you use. List boxes also include scroll bars when the number of items in the list exceeds the visible area of the control.

Arrow keys also provide support for selection and scrolling a list box. In addition, list boxes include support for keyboard selection using text keys. When the user presses a text key, the list navigates and selects the matching item in the list, scrolling the list if necessary to keep the user's selection visible. Subsequent key presses continue the matching process. Some list boxes support sequential matches based on timing; each time the user presses a key, the control matches the next character in a word if the user presses the key within the system's time-out setting. If the time-out elapses, the control is reset to matching based on the first character. Other list box controls, such as combo boxes and drop-down combo boxes, do sequential character matching based on the characters typed into the text box component of the control. These controls may be preferable because they do not require the user to master the timing sequence. However, they do take up more space and potentially allow the user to type in entries that do not exist in the list box.

When the list is scrolled to the beginning or end of data, disable the corresponding scroll bar arrow button. If all items in the list are visible, disable both scroll arrows. If the list box never includes more items that can be shown in the list box, so that the user will not need to scroll the list, you may remove

the scroll bar.

When incorporating a list box into a window’s design, consider supporting both command (Cut, Copy, and Paste) and direct manipulation (drag and drop) transfers for the list box. For example, if the list displays icons or values that the user can move or copy to other locations, such as another list box, support transfer operations for the list. The list view control automatically supports this; however, the system provides support for you to enable this for other list boxes as well.

For more information about disabling scroll bar arrows, see Windows.

List boxes can be classified by how they display a list and by the type of selection they support.

## Single Selection List Boxes

A *single selection list box* is designed for the selection of only one item in a list. Therefore, the control provides a mutually exclusive operation similar to a group of option buttons, except that a list box can more efficiently handle a large number of items.

Define a single selection list box to be tall enough to show at least three to eight choices, as shown in Figure 7.13 — depending on the design constraints of where the list box is used. Always include a vertical scroll bar. If all the items in the list are visible, then follow the window scroll bar guidelines for disabling the scroll arrows and enlarging the scroll box to fill the scroll bar shaft.

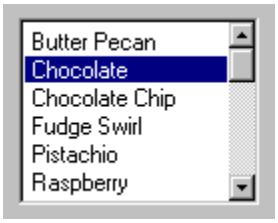



**Figure 7.13 A single selection list box**

The currently selected item in a single selection list box is highlighted using selection appearance.

The user can select an entry in a single selection list box by clicking on it with mouse button 1 (for pens, tapping). This also sets the input focus on that item in the list. Because this type of list box supports only single selection, when the user chooses another entry any other selected item in the list becomes unselected. The scroll bar in the list box allows the mouse user to scroll through the list of entries, following the interaction defined for scroll bars.

For more information about the interaction techniques of scroll bars, see Windows.

The keyboard interface uses navigation keys, such as the arrow keys, HOME, END, PAGE UP, and PAGE DOWN. It also uses text keys, with matches based on timing; for example, when the user presses a text key, an entry matching that character scrolls to the top of the list and becomes selected. These keys not only navigate to an entry in the list, but also select it. If no item in the list is currently selected, when the user chooses a list navigation key, the first item in the list that corresponds to that key is selected. For example, if the user presses the DOWN ARROW key, the first entry in the list is selected, instead of navigating to the second item in the list.

If the choices in the list box represent values for the property of a selection, then make the current value visible and highlighted when displaying the list. If the list box reflects mixed values for a multiple selection, then no entry in the list should be selected.

## Drop-down List Boxes

Like a single selection list box, a *drop-down list box* provides for the selection of a single item from a list of items; the difference is that the list is displayed upon demand. In its closed state, the control displays the current value for the control. The user opens the list to change the value. Figure 7.14 shows the drop-down list box in its closed and opened state.

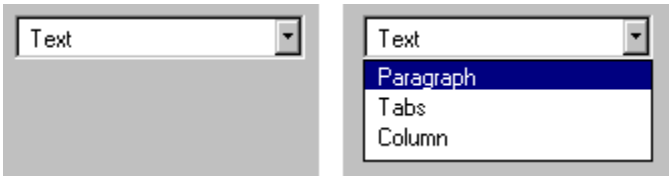



**Figure 7.14 A drop-down list box (closed and opened state)**

While drop-down list boxes are an effective way to conserve space and reduce clutter, they require more user interaction for browsing and selecting an item than a single selection list box.

Make the width of a closed drop-down list box a few spaces larger than the average width of the items in its list. The open list component of the control should be tall enough to show three to eight items, following the same conventions of a single selection list box. The width of the list should be wide enough not only to display the choices in the list, but also to allow the user to drag directly into the list.

The interface for drop-down list boxes is similar to that for menus. For example, the user can press the mouse button on the current setting portion of the control or on the control's menu button to display the list. Choosing an item in the list automatically closes the list.

If the user navigates to the control using an access key, the TAB key or arrow keys, an UP ARROW or DOWN ARROW, or ALT+UP ARROW or ALT+DOWN ARROW displays the list. Arrow keys or text keys navigate and select items in the list. If the user presses ALT+UP ARROW, ALT+DOWN ARROW, a navigation key, or an access key to move to another control, the list automatically closes. When the list is closed, preserve any selection made while the list was open. The ESC key also closes the list.

If the choices in a drop-down list represent values for the property of a multiple selection and the values for that property are mixed, then display no value in the current setting component of the control.

## Extended and Multiple Selection List Boxes

Although most list boxes are single selection lists, some contexts require the user to choose more than one item. *Extended selection list boxes* and *multiple selection list boxes* support this functionality.

Extended and multiple selection list boxes follow the same conventions for height and width as single selection list boxes. The height should display no less than three items and generally no more than eight, unless the size of the list varies with the size of the window. Base the width of the box on the average width of the entries in the list.

Extended selection list boxes support conventional navigation, and contiguous and disjoint selection techniques. That is, extended selection list boxes are optimized for selecting a single item or a single range, while still providing for disjoint selections.

For more information about contiguous and disjoint selection techniques, see General Interaction Techniques.

When you want to support user selection of several disjoint entries from a list, but an extended selection list box is too cumbersome, you can define a multiple selection list box. Whereas extended selection list boxes are optimized for individual item or range selection, multiple selection list boxes are optimized for independent selection. However, because simple multiple selection list boxes are not visually distinct from extended selection list boxes, consider designing them to appear similar to a scrollable list of check boxes, as shown in Figure 7.15. This requires providing your own graphics for the items in the list (using the owner-drawn list box style). This appearance helps the user to distinguish the difference in the interface of the list box with a familiar convention. It also serves to differentiate keyboard navigation from the state of a choice. Because the check box controls are nested, you use the flat appearance style for the check boxes. You may also create this kind of a list box using a list view control.

For more information about the flat appearance style for controls in a list box, see Visual Design.


**Figure 7.15 A multiple selection list box**

# List View Controls

A *list view control* is a special extended selection list box that displays a collection of items, each item consisting of an icon and a label. List view controls can display content in four different views.

| View | Description |
|---|---|
| Icon | Each item appears as a full-sized icon with a label below it. The user can drag the icons to any location within the view. |
| Small Icon | Each item appears as a small icon with its label to the right. The user can drag the icons to any location within the view. |
| List | Each item appears as a small icon with its label to the right. The icons appear in a columnar, sorted layout. |
| Report | Each item appears as a line in a multicolumn format with the leftmost column including the icon and its label. The subsequent columns contain information supplied by the application displaying the list view control. |

The control also supports options for alignment of icons, selection of icons, sorting of icons, and editing of the icon’s labels. It also supports drag and drop interaction.

Use this control where the representation of objects as icons is appropriate. In addition, provide pop-up menus on the icons displayed in the views. This provides a consistent paradigm for how the user interacts with icons elsewhere in the Windows interface.

Selection and navigation in this control work similarly to that in folder windows. For example, clicking on an icon selects it. After selecting the icon, the user can use extended selection techniques, including region selection, for contiguous or disjoint selections. Arrow keys and text keys (time-out based matching) support keyboard navigation and selection.

As an option, the standard control also supports the display of graphics that can be used to represent state information. For example, you can use this functionality to include check boxes next to items in a list.

## Tree View Controls

A *tree view control* is a special list box control that displays a set of objects as an indented outline based on their logical hierarchical relationship. The control includes buttons that allow the outline to be expanded and collapsed, as shown in Figure 7.16. You can use a tree view control to display the relationship between a set of containers or other hierarchical elements.

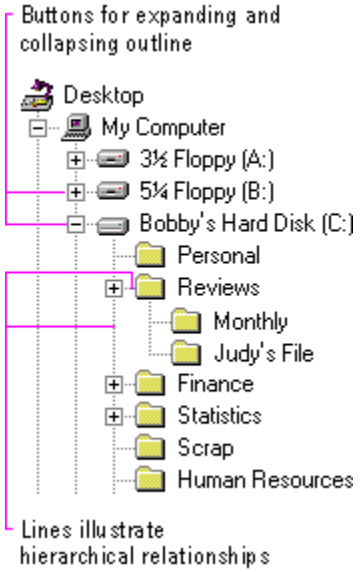



**Figure 7.16 A tree view control**

You can optionally include icons with the text label of each item in the tree. Different icons can be displayed when the user expands or collapses the item in the tree. In addition, you can also include a graphic, such as a check box, that can be used to reflect state information about the item.

The control also supports drawing lines that define the hierarchical relationship of the items in the list and buttons for expanding and collapsing the outline. It is best to include these features (even though they are optional) because they make it easier for the user to interpret the outline.

Arrow keys provide keyboard support for navigation through the control; the user presses UP ARROW and DOWN ARROW to move between items and LEFT ARROW and RIGHT ARROW to move along a particular branch of the outline. Pressing RIGHT ARROW can also expand the outline at a branch if it is not currently displayed. Text keys can also be used to navigate and select items in the list, using the matching technique based on timing.

When you use this control in a dialog box, if you use the ENTER key or use double-clicking to carry out the default command for an item in the list, make certain that the default command button in your dialog box matches. For example, if you use double-clicking an entry in the outline to display the item's properties, then define a Properties button to be the default command button in the dialogbox when the tree view control has the input focus.

# Text Fields

Windows includes a number of controls that facilitate the display, entry, or editing of a text value. Some of these controls combine a basic text-entry field with other types of controls.

Text fields do not include labels as a part of the control. However, you can add one using a static text field. Including a label helps identify the purpose of a text field and provides a means of indicating when the field is disabled. Use sentence capitalization for multiple word labels. You can also define access keys for the text label to provide keyboard access to the text field. When using a static text label, define keyboard access to move the input focus to the text field with which the label is associated rather than the static text field itself. You can also support keyboard navigation to text fields by using the TAB key (and, optionally, arrow keys).

For more information about static text fields, see Static Text Fields.

When using a text field for input of a restricted set of possible values, for example, a field where only numbers are appropriate, validate user input immediately, either by ignoring inappropriate characters or by providing feedback indicating that the value is invalid or both.

For more information about validation of input, see Secondary Windows.

## Text Boxes

A *text box* (also referred to as an edit control) is a rectangular control where the user enters or edits text, as shown in Figure 7.17. It can be defined to support a single line or multiple lines of text. The outline border of the control is optional, although the border is typically included when displaying the control in a toolbar or a secondary window.


**Figure 7.17 A standard text box**

The standard text box control provides basic text input and editing support. Editing includes the insertion or deletion of characters and the option of text wrapping. Although individual font or paragraph properties are not supported, the entire control can support a specific font setting.

You can also use text boxes to display read-only text that is not editable, but still selectable. When setting this option with the standard control, the system automatically changes the background color of the field to indicate to the user the difference in behavior.

A text box supports standard interactive techniques for navigation and contiguous selection. Horizontal scrolling is available for single line text boxes, and horizontal and vertical scroll bars are supported for multiple line text boxes.

You can limit the number of characters accepted as input for a text box to whatever is appropriate for the context. In addition, you can support *auto-exit* for text boxes defined for fixed-length input; that is, as soon as the last character is typed in the text box, the focus moves to the next control. For example, you can define a five-character auto-exit text box to facilitate the entry of zip code, or three two-character auto-exit text boxes to support the entry of a date. Use auto-exit text boxes sparingly; the automatic shift of focus can surprise the user. They are best limited to situations involving extensive data entry.

## Rich-Text Boxes

A *rich-text box*, as shown in Figure 7.18, provides the same basic text editing support as a standard text box. In addition, a rich-text box supports font properties, such as typeface, size, color, bold, and italic format, for each character and paragraph format property, such as alignment, tabs, indents, and numbering. The control also supports printing of its content and embedding of OLE objects.

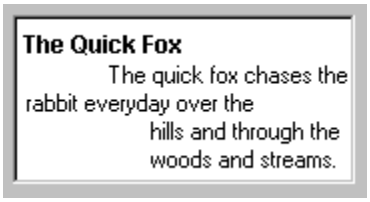



**Figure 7.18 A rich-text box**

## Combo Boxes

A *combo box* is a control that combines a text box with a list box, as shown in Figure 7.19. This allows the user to type in an entry or choose one from the list.

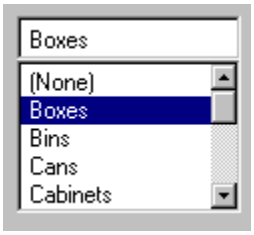



**Figure 7.19 A combo box**

The text box and its associated list box have a dependent relationship. As text is typed into the text box, the list scrolls to the nearest match. In addition, when the user selects an item in the list box, it automatically uses that entry to replace the content of the text box and selects the text.

The interface for the control follows the conventions supported for each component, except that the UP ARROW and DOWN ARROW keys move only in the list box. LEFT ARROW and RIGHT ARROW keys operate solely in the text box.

## Drop-down Combo Boxes

A *drop-down combo box*, as shown in Figure 7.20, combines the characteristics of a text box with a drop-down list box. A drop-down combo box is more compact than a regular combo box; it can be used to conserve space, but requires additional user interaction required to display the list.

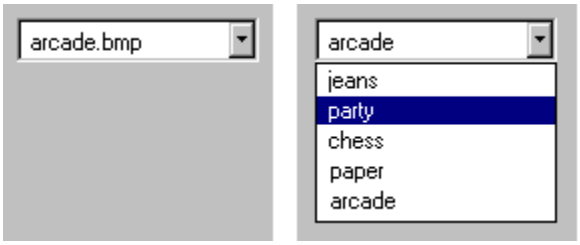



**Figure 7.20 A drop-down combo box (closed and opened state)**

The closed state of a drop-down combo box is similar to that of a drop-down list, except that the text box is interactive. When the user clicks the control's menu button the list is opened. Clicking the menu button a second time, choosing an item in the list, or clicking another control closes the list.

Provide a static text field label for the control and assign an access key. Use the access key so the user can navigate to the control. You can also support the TAB key or arrow keys for navigation to the control. When the control has the input focus, when the user presses the UP ARROW or DOWN ARROW or ALT+UP ARROW or ALT+DOWN ARROW key, the list is displayed.

When the control has the input focus, pressing a navigation key, such as the TAB key, or an access key or ALT+UP ARROW or ALT+DOWN ARROW to navigate to another control closes the list. When the list is closed, preserve any selection made while the list was open, unless the user presses a Cancel command button. The ESC key also closes the list.

When the list is displayed, the interdependent relationship between the text box and list is the same as it is for standard combo boxes when the user types text into the text box. When the user chooses an item in the list, the interaction is the same as for drop-down lists — the selected item becomes the entry in the text box.

## Spin Boxes

*Spin boxes* are text boxes that accept a limited set of discrete ordered input values that make up a circular loop. A spin box is a combination of a text box and a special control that incorporates a pair of buttons (also known as an up-down control), as shown in Figure 7.21.

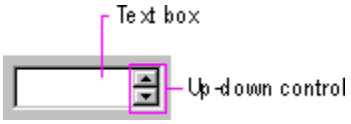



**Figure 7.21 A spin box**

When the user clicks on the text box or the buttons, the input focus is set to the text box component of the control. The user can type a text value directly into the control or use the buttons to increment or decrement the value. The unit of change depends on what you define the control to represent.

Use caution when using the control in situations where the meaning of the buttons may be ambiguous. For example, with numeric values, such as dates, it may not be clear whether the top button increments the date or changes to the previous date. Define the top button to increase the value by one unit and the bottom button to decrease the value by one unit. Typically, wrap around at either end of the set of values. You may need to provide some additional information to communicate how the buttons apply.

By including a static text field as a label for the spin box and defining an associated access key, you can provide direct keyboard access to the control. You can also support keyboard access using the TAB key (or, optionally, arrow keys). Once the control has the input focus, the user can change the value by pressing UP ARROW or DOWN ARROW.

You can also use a single set of spin box buttons to edit a sequence of related text boxes, for example, time as expressed in hours, minutes, and seconds. The buttons affect only the text box that currently has the input focus.

## Static Text Fields

You can use static text fields to present read-only text information. Unlike read-only text box controls, the text is not selectable. However, your application can still alter read-only static text to reflect a change in state. For example, you can use static text to display the current directory path or the status information, such as page number, key states, or time and date. Figure 7.22 illustrates a static text field.


**Figure 7.22 A static text field**

You can also use static text fields to provide labels or descriptive information for other controls. Using static text fields as labels for other controls allows you to provide access-key activation for the control with which it is associated. Make certain that the input focus moves to its associated control and not to the static field. Also remember to include a colon at the end of the text. Not only does this help communicate that the text represents the label for a control, it is also used by screen review utilities.

For more information about the layout of static text fields, see Visual Design. For information about the use of static text fields as labels and screen review utilities, see Special Design Considerations.

# Shortcut Key Input Controls

A *shortcut key input control* (also known as a hot key control) is a special kind of text box to support user input of a key or key combination to define a shortcut key assignment. Use it when you provide an interface for the user to customize shortcut keys supported by your application. Because shortcut keys carry out a command directly, they provide a more efficient interface for common or frequently used actions.

For more information about the use of shortcut keys, see Input Basics.

The control allows you to define invalid keys or key combinations to ensure valid user input; the control will only access valid keys. You also supply a default modifier to use when the user enters an invalid key. The control displays the valid key or key combination including any modifier keys.

When the user clicks a shortcut key input control, the input focus is set to the control. Like most text boxes, the control does not include its own label, so use a static text field to provide a label and assign an appropriate access key. You can also support the TAB key to provide keyboard access to the control.

## Other General Controls

The system also provides support for controls designed to organize other controls and controls for special types of interfaces.

## Group Boxes

A *group box* is a special control you can use to organize a set of controls. A group box is a rectangular frame with an optional label that surrounds a set of controls, as shown in Figure 7.23. Group boxes generally do not directly process any input. However, you can provide navigational access to items in the group using the TAB key or by assigning an access key to the group label.

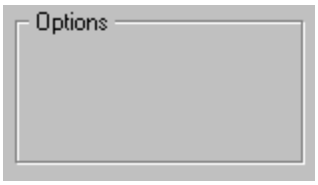



**Figure 7.23 A group box**

You can make the label for controls that you place in a group box relative to the group box's label. For example, a group labeled Alignment can have option buttons labeled Left, Right, and Center. Use sentence capitalization for a multiple word label.

## Column Headings

Using a *column heading* control, also known as a header control, you can display a heading above columns of text or numbers. You can divide the control into two or more parts to provide headings for multiple columns, as shown in Figure 7.24. The list view control also provides support for a column heading control.

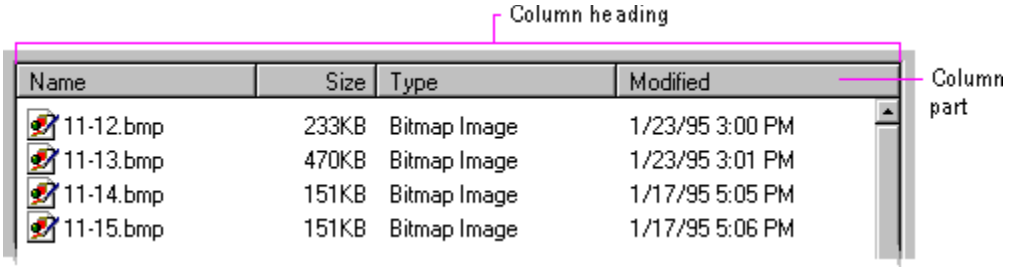



**Figure 7.24 A column heading divided into four parts**

Each header part label can include text and a graphic image. Use the graphic image to show information such as the sort direction. You can align the title elements left, right, or centered.

You can configure each part to behave like a command button to support a specific function when the user clicks on it. For example, consider supporting sorting the list by clicking on a particular header part. Also, you can support clicking on the part with button 2 to display a pop-up menu containing specific commands, such as Sort Ascending and Sort Descending.

The control also supports the user dragging on the divisions that separate header parts to set the width of each column. As an option, you can support double-clicking on a division as a shortcut to a command that applies to formatting the column, such as automatically sizing the column to the largest value in that column.

## Tabs

A *tab* control is analogous to a divider in a file cabinet or notebook, as shown in Figure 7.25. You can use this control to define multiple logical pages or sections of information within the same window.

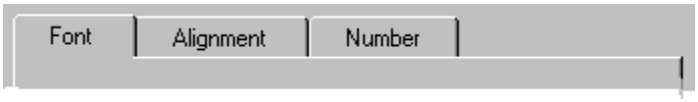



**Figure 7.25 A tab control**

Tab labels can include text or graphic information, or both. Usually, the control automatically sizes the tab to the size of its label; however, you can define your tabs to have a fixed width. Use the system font for the text labels of your tabs and use the same capitalization for multiple word labels as you use for menus and command buttons (in English versions, book title capitalization). If you use only graphics as your tab label, support tooltips for your tabs.

By default, a tab control displays only one row of tabs. While the control supports multiple rows or scrolling a single row of tabs, avoid these alternatives because they add complexity to the interface by making it harder to read and access a particular tab. You may want to consider alternatives such as separating the tabbed pages into sets and using another control to move between the sets. However, if scrolling the tabs seems appropriate, follow the conventions documented in this guide.

When the user clicks a tab with mouse button 1, the input focus moves and switches to that tab. When a tab has the input focus, LEFT ARROW or RIGHT ARROW keys move between tabs. CTRL+TAB also switches between tabs. Optionally, you can also define access keys for navigating between tabs. If the user switches pages using the tab, you can place the input focus on the particular control on that page. If there is no appropriate control or field in which to place the tab, leave the input focus on the tab itself.

# Property Sheet Controls

A *property sheet control* provides the basic framework for defining a property sheet. It provides the common controls used in a property sheet and accepts modeless dialog box layout definitions to automatically create tabbed property pages.

The property sheet control also includes support for creating wizards. Wizards are a special form of user assistance that guide the user through a sequence of steps in a specific operation or process. When using the control as a wizard, tabs are not included, and the standard OK, Cancel, and Apply buttons are replaced with a Back, Next, or Finish button, and a Cancel button.

For more information about property sheets, see Secondary Windows. For more information about wizards, see User Assistance.

# Scroll Bars

*Scroll bars* are horizontal or vertical scrolling controls you can use to create scrollable areas other than on the window frame or list box where they can be automatically included. Use scroll bar controls only for supporting scrolling contexts. For contexts where you want to provide an interface for setting or adjusting values, use a slider or other control, such as a spin box. Because scroll bars are designed for scrolling information, using a scroll bar to set values may confuse the user as to the purpose or interaction of the control.

When using scroll bar controls, follow the recommended conventions for disabling the scroll bar arrows. Disable a scroll bar arrow button when the user scrolls the information to the beginning or end of the data, unless the structure permits the user to scroll beyond the data. For more information about scroll bar conventions, see Windows.

While scroll bar controls can support the input focus, avoid defining this type of interface. Instead, define the keyboard interface of your scrollable area so that it can scroll without requiring the user to move the input focus to a scroll bar. This makes your scrolling interface more consistent with the user interaction for window and list box scroll bars.

## Sliders

Use a slider for setting or adjusting values on a continuous range of values, such as volume or brightness. A *slider* is a control, sometimes called a trackbar control, that consists of a bar that defines the extent or range of the adjustment, and an indicator that both shows the current value for the control and provides the means for changing the value, as shown in Figure 7.26.

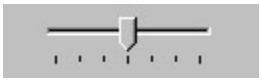


**Figure 7.26 A slider**

Because a slider does not include its own label, use a static text field to create one. You can also add text and graphics to the control to help the user interpret the scale and range of the control.

Sliders support a number of options. You can set the slider orientation as vertical or horizontal, define the length and height of the slide indicator and the slide bar component, define the increments of the slider, and whether to display tick marks for the control.

The user moves the slide indicator by dragging to a particular location or clicking in the hot zone area of the bar, which moves the slide indicator directly to that location. To provide keyboard interaction, support the TAB key and define an access key for the static text field you use for its label. When the control has the input focus, arrow keys can be used to move the slide indicator in the respective direction represented by the key.

## Progress Indicators

A *progress indicator* is a control, also known as a progress bar control, you can use to show the percentage of completion of a lengthy operation. It consists of a rectangular bar that “fills” from left to right, as shown in Figure 7.27.

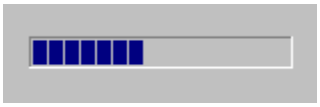


**Figure 7.27 A progress indicator**

Because a progress indicator only displays information, it is typically noninteractive. However, it may be useful to add static text or other information to help communicate the purpose of the progress indicator. If you do include text, place it outside of the progress indicator control.

Use the control as feedback for long operations or background processes as a supplement to changing the pointer. The control provides more visual feedback to the user about the progress of the process. You can also use the control to reflect the progression of a background process, leaving the pointer’s image to reflect interactivity for foreground activities. When determining whether to use a progress indicator in message box or status bar, consider how modal the operation or process the progress indicator represents.

For more information about message boxes, see Secondary Windows. For more information about status bars, see Toolbars and Status Bars.

## Tooltip Controls

A tooltip control provides the basic functionality of a tooltip. A tooltip is a small pop-up window that includes descriptive text displayed when the user moves the pointer over a control, as shown in Figure 7.28. The tooltip appears after a short time-out and is automatically removed when the user clicks the control or moves the pointer off the control.

For more information about the use of tooltips, see User Assistance. For more information about the use of tooltips in toolbars, see Toolbars and Status Bars.


**Figure 7.28 A tooltip control**

The system displays a tooltip control at the lower right of the pointer, but automatically adjusts the tooltip to avoid displaying it offscreen. However for text boxes, the tooltip should be displayed centered under the control it identifies. The control supports an option to support this behavior.

## Wells

A *well* is a special field similar to a group of option buttons, but facilitates user selection of graphic values such as a color, pattern, or images, as shown in Figure 7.29. This control is not currently provided by the system; however, its purpose and interaction guidelines are described here to provide a consistent interface.

**Figure 7.29 A well control for selection colors**

Like option buttons, use well controls for values that have two or more choices and group the choices to form a logical arrangement. When the control is interactive, use the same border pattern as a check box or text box. When the user chooses a particular value in the group, indicate the set value with a special selection border drawn around the edge of the control.

For more information about how to display well controls, see Visual Design.

Follow the same interaction techniques as option buttons. When the user clicks a well in the group the value is set to that choice. Provide a group box or static text to label the group and define an access key for that label and supporting the TAB key to navigate to a group. Use arrow keys to move between values in the group.

## Pen-Specific Controls

When the user installs a pen input device, single line text boxes and combo boxes automatically display a writing tool button described in General Interaction Techniques. In addition, the system provides special controls for supporting pen input.

## Boxed Edit Controls

A *boxed edit* control provides the user with a discrete area for entering characters. It looks and operates similarly to a writing tool window without some of the writing tool window's buttons, as shown in Figure 7.30.

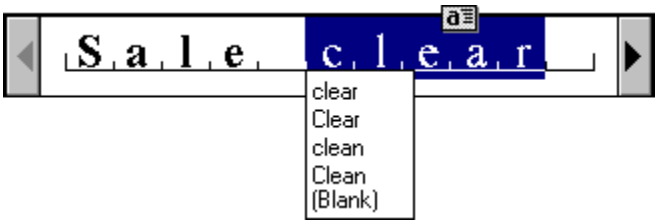



**Figure 7.30 A single line boxed edit control**

Both single and multiple line boxed edit controls are supported. Figure 7.31 shows a multiple line boxed edit control.

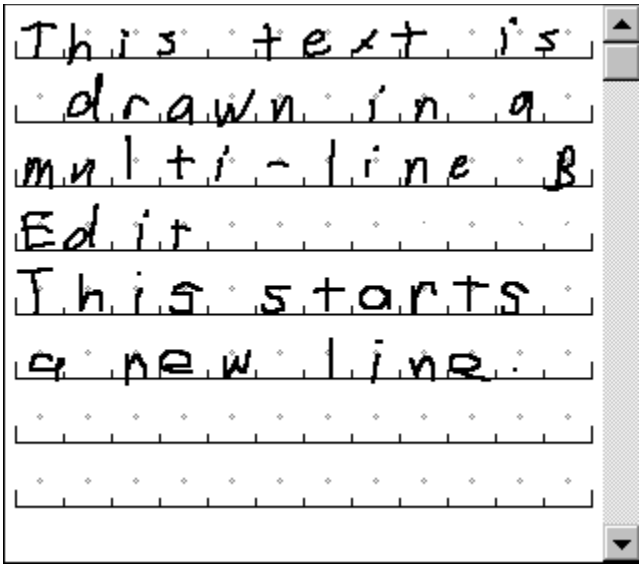



**Figure 7.31 A multiple line boxed edit control**

Like the writing tool window, these controls provide a pen selection handle for selection of text and an action handle for operations on a selection. They also provide easy correction by overwriting and selecting alternative choices.

## Ink Edit Controls

The *ink edit* is a pen control in which the user can create and edit lines drawn as ink; no recognition occurs here. It is a drawing area designed for ink input, as shown in Figure 7.32.

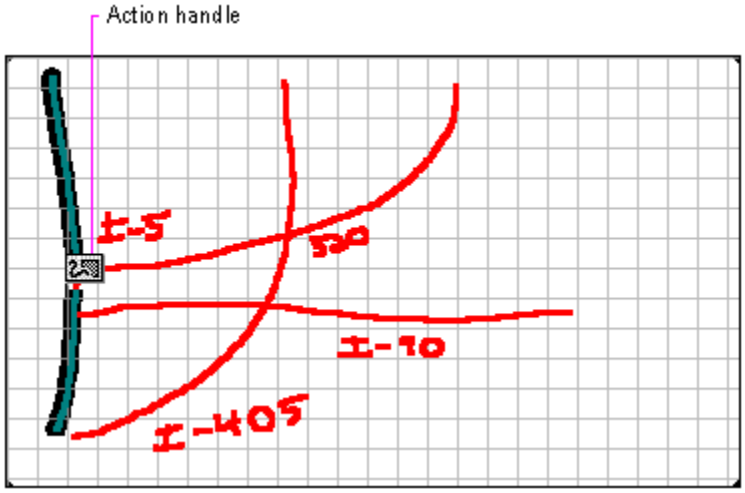



**Figure 7.32 An ink edit control**

The control provides support for an optional grid, optional scroll bars, and optional display of a frame border. Selection is supported using tapping to select a particular stroke; lasso-tapping is also supported for selecting single or multiple strokes. After the user makes a selection, an action handle is displayed. Tapping on the action handle displays a pop-up menu that includes commands for Undo, Cut, Copy, Paste, Delete, Use Eraser, Resize, What's This?, and Properties. Choosing the Properties command displays a property sheet associated with the selection — this allows the user to change the stroke width and color.

If you use an ink edit control, you may also want to include some controls for special functions. For example, a good addition is an Eraser button, as shown in Figure 7.33.

**Figure 7.33 The eraser toolbar button**

Implement the Eraser button to operate as a "spring-loaded" mode; that is, choosing the button causes the pen to act as an eraser while the user presses the pen to the screen. As soon as it is lifted, the pen reverts to its drawing mode.

## Toolbars and Status Bars

Like menu bars, toolbars and status bar are special interface constructs for managing sets of controls. A *toolbar* is a panel that contains a set of controls, as shown in Figure 7.34, designed to provide quick access to specific commands or options. Specialized toolbars are sometimes called ribbons, tool boxes, and palettes.

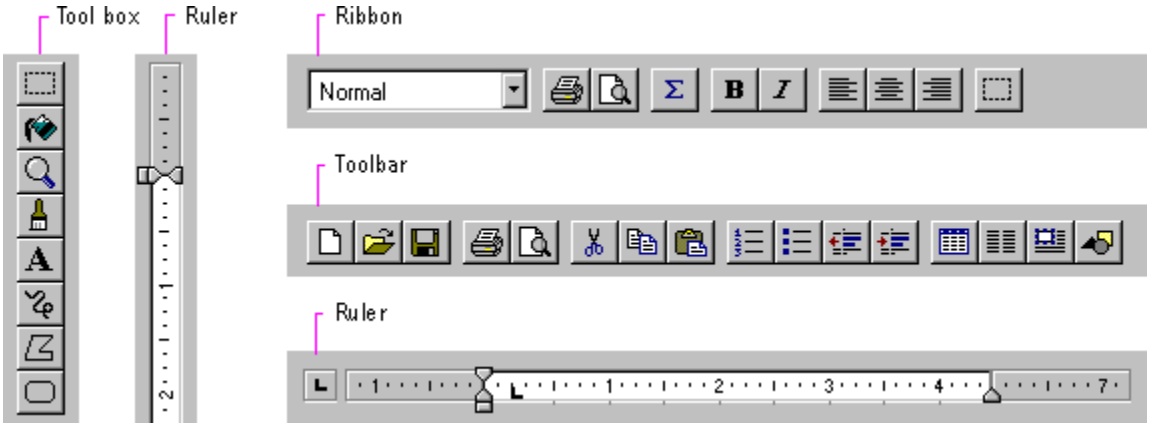



**Figure 7.34 Examples of toolbars**

A *status bar*, shown in Figure 7.35, is a special area within a window, typically the bottom, that displays information about the current state of what is being viewed in the window or any other contextual information, such as keyboard state. You can also use the status bar to provide descriptive messages about a selected menu or toolbar button. Like a toolbar, a status bar can contain controls; however, typically include read-only or noninteractive information.

For more information about status bar messages, see User Assistance.

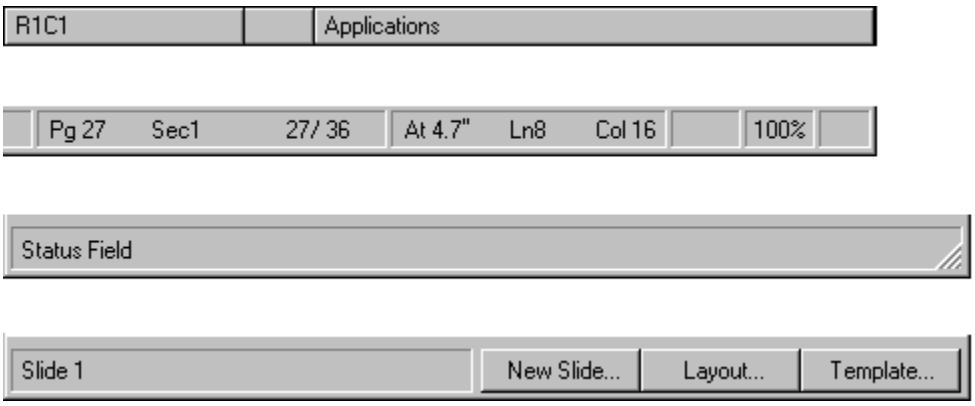



**Figure 7.35 Examples of status bars**

## Interaction with Controls in Toolbars and Status Bars

The user can access the controls included in a toolbar or status bar with the mouse or pen through the usual means of interaction for those controls. You can provide keyboard access using either shortcut keys or access keys. If a control in a toolbar or status bar does not have a text label, access keys may not be as effective. Furthermore, if a particular access key is already in use in the primary window, it may not be available for accessing the control in the toolbar. For example, if the menu bar of the primary window is already using a particular access key, then the menu bar receives the key event.

When the user interacts with controls in a toolbar or status bar that reflect properties, any change is directly applied to the current selection. For example, if a button in a toolbar changes the property of text to bold, choosing that button immediately changes the text to bold; no further confirmation or transaction action is required. The only exception is if the control, such as a button, requires additional input from the user; then the effect may not be realized until the user provides the information for those parameters. An example of such an exception would be the selection of an object or a set of input values through a dialog box.

Always provide a tooltip for controls you include in a toolbar or status bar that do not have a text label. The system provides support for tooltips in the standard toolbar control and a tooltip control for use in other contexts.

## Support for User Options

To provide maximum flexibility for users and their tasks, design your toolbars and status bars to be user configurable. Providing the user with the option to display or hide toolbars and status bars is one way to do this. You can also include options that allow the user to change or rearrange the elements included in toolbars and status bars.

Provide toolbar buttons in at least two sizes: 24 by 22 and 32 by 30 pixels. To fit a graphic label in these button sizes, design the images no larger than 16 by 16 and 24 by 24 pixels, respectively. In addition, support the user's the option to change between sizes by providing a property sheet for the toolbar (or status bar).

For more information about designing toolbar buttons, see Visual Design.

Consider also making the location of toolbars user adjustable. While toolbars are typically *docked* by default — aligned to the edge of a window or pane to which they apply — design your toolbars to be moveable so that the user can dock them along another edge or display them as a palette window.

For more information about palette windows, see Secondary Windows.

To undock a toolbar from its present location, the user must be able to click anywhere in the "blank" area of the toolbar and drag it to its new location. If the new location is within the hot zone of an edge, your application should dock the toolbar at the new edge when the user releases the mouse button. If the new location is not within the hot zone of an edge, redisplay the toolbar in a palette window. To redock the window with an edge, the user drags the window by its title bar until the pointer enters the hot zone of an edge. Return the toolbar to a docked state when the user releases the mouse button.

As the user drags the toolbar, provide visual feedback, such as a dotted outline of the toolbar. When the user moves the pointer into a hot zone of a dockable location, display the outline in its docked configuration to provide a cue to the user about what will happen when the drag operation is complete. You can also support user options such as resizing the toolbar by dragging its border or docking multiple toolbars side by side, reconfiguring their arrangement and size as necessary.

When supporting toolbar and status bar configuration options, avoid including controls whose functionality is not available elsewhere in the interface. In addition, always preserve the current position and size, and other state information, of toolbar and status bar configuration so that they can be restored to their state when the user reopens the window.

## Toolbar and Status Bar Controls

The system includes toolbar and status bar controls that you can use to implement these interfaces in your applications. The toolbar control supports docking and windowing functionality. It also supports a dialog box for allowing the user to customize the toolbar. You define whether the customization features are available to the user and what features the user can customize. The system also supports creation of desktop toolbars. For more information about desktop toolbars, see Integrating with the System.

The standard status bar control also includes the option of including a size grip control for sizing the window, described in Windows. When the status bar size grip is displayed, if the window displays a size grip at the junction of the horizontal and vertical scroll bars of a window, that grip should be hidden so that it does not appear in both locations at the same time. Similarly, if the user hides the status bar, restore the size grip at the corner of the scroll bars.

## Common Toolbar Buttons

Table 7.5 illustrates the button images that you can use for common functions.

**Table 7.5 Common Toolbar Buttons**

| 16 x 16 button | 24 x 24 button | Function |
|---|---|---|
|  |  | New |
|  |  | Open |
|  |  | Save |
|  |  | Print |
|  |  | Print Preview |
|  |  | Undo |
|  |  | Redo |
|  |  | Cut |
|  |  | Copy |
|  |  | Paste |
|  |  | Delete |
|  |  | Find |
|  |  | Replace |
|  |  | Properties |

| | | |
|---|---|---|
|  |  | Bold |
|  |  | Italic |
|  |  | Underline |
|  |  | What's This? (context-sensitive Help mode) |
|  |  | Show Help Topics |
|  |  | Open parent folder |
|  |  | View as large icons |
|  |  | View as small icons |
|  |  | View as list |
|  |  | View as details |
|  |  | Region selection tool |
|  |  | Writing tool (pen) |
|  |  | Eraser tool (pen) |

Use these images only for the function described. Consistent use of these common tool images allows the user to transfer their learning and skills from product to product. If you use one of the standard images for a different function, you may confuse the user. When designing your own toolbar buttons, follow the conventions supported by the standard system controls.

For more information about the design of toolbar buttons, see Visual Design.

# Introduction

Most primary windows require a set of secondary windows to support and supplement a user's activities in the primary windows. Secondary windows are similar to primary windows but differ in some fundamental aspects. This topic covers the common uses of secondary windows, such as property sheets, dialog boxes, palette windows, and message boxes.

## Characteristics of Secondary Windows

Although secondary windows share some characteristics with primary windows, they also differ from primary windows in their behavior and use. For example, secondary windows should not appear on the taskbar. Secondary windows obtain or display supplemental information which is often related to the objects that appear in a primary window.

## Appearance and Behavior

A typical secondary window, as shown in Figure 8.1, includes a title bar and frame; a user can move it by dragging its title bar. However, a secondary window does not include Maximize and Minimize buttons because these sizing operations typically do not apply to a secondary window. A Close button can be included to dismiss the window. The title text is a label that describes the purpose of the window; the content of the label depends on the use of the window. The title bar does not include icons.

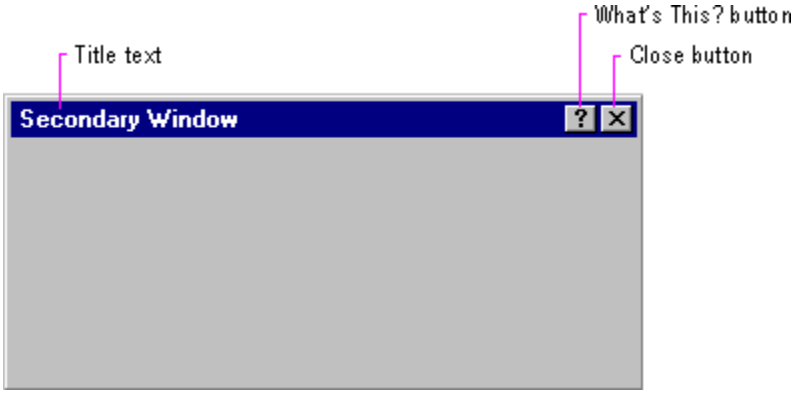



**Figure 8.1 A secondary window**

You can include status information in secondary windows, but avoid including a status bar control used in primary windows.

Like a primary window, a secondary window includes a pop-up menu with commands that apply to the window. A user can access the pop-up menu for the window using the same interaction techniques as primary windows.

A secondary window can also include a What's This? button in its title bar. This button allows a user to display context-sensitive Help information about the components displayed in the window.

## Interaction with Other Windows

Secondary windows that are displayed because of commands chosen within a primary window depend on the state of the primary window; that is, when the primary window is closed or minimized, its secondary windows are also closed or hidden. When the user reopens or restores the primary window, restore the secondary windows to their former positions and states. However, if opening a secondary window is the result of an action outside of the object's primary window — for example, if the user chooses the Properties command on an icon in a folder or on the desktop — then the property sheet window is independent and appears as a peer with any primary windows, though it should not appear in the taskbar.

When the user opens or switches to a secondary window, it is activated or deactivated like any other window. With the mouse or pen, the user activates a secondary window in the same way as a primary window. With the keyboard, the ALT+F6 key combination switches between a secondary window and its primary window, or other peer secondary windows that are related to its primary window. A secondary window must be modeless to support this form of switching.

When the user activates a primary window, bringing it to the top of the window Z order, all of its dependent secondary windows also come to the top, maintaining their same respective order. Similarly, activating a dependent secondary window brings its primary window and related peer windows to the top.

A dependent secondary window always appears on top of its associated primary window, layered with any related window that is a peer secondary window. When activated, the secondary window appears on top of its peers. When a peer is activated, the secondary window appears on top of its primary window, but behind the newly activated secondary window that is a peer.

You can design a secondary window to always appear at the top of its peer secondary windows. Typically, you should use this technique only for palette windows and, even in this situation, make this feature configurable by the user by providing an Always On Top property setting for the window. If you support this technique for multiple secondary windows, then the windows are managed in their own Z order within the collection of windows of which they are a part.

Avoid having a secondary window with the Always On Top behavior appear on top of another application's primary window (or any of the other application's dependent secondary windows) when the user activates a window of that application, unless the Always On Top window can also be applied to that application's windows.

When the user chooses a command that opens a secondary window, use the context of the operation to determine how to present information in that window. In property sheets, for example, set the values of the properties in that window to represent the selection.

In general, display the window in the same state as the user last accessed it. For example, an Open dialog box should preserve the current directory setting between the openings of a window. Similarly, if you use tabbed pages for navigating through information in a secondary window, display the last page the user was viewing when the user closed the window. This makes it easier for the user to repeat an operation that is associated with the window. It also provides more stability in the interface.

However, if a command or task implies or requires that the user begin a process in a particular sequence or state, such as with a wizard window, you should present the secondary window using a fixed or consistent presentation. For example, entering a record into a database may require the user to enter the data in a particular sequence. Therefore, it may be more appropriate to present the input window always displaying the first entry field.

## Unfolding Secondary Windows

Except for palette windows, avoid defining secondary windows to be resizable because their purpose is to provide concise, predefined information. However, you can use an unfold button to expand a window to reveal additional options as a form of progressive disclosure. An *unfold button* is a command button with a label that includes two “greater than” characters (>>). When the user chooses the button, the secondary window expands to its alternative fixed size. As an option, you can use the button to “refold” the additional part of the window.

## Cascading Secondary Windows

You can also provide the user access to additional options by including a command button that opens another secondary window. If the resulting window is independent in its operation, close the secondary window from which the user opened it and display only the new window. However, if the intent of the subsequent window is to obtain information for a field in the original secondary window, then the original should remain displayed and the dependent window should appear on top, offset slightly to the right and below the original secondary window. When using this latter method, limit the number of secondary windows to a single level to avoid creating a cluttered cascading chain of hierarchical windows.

## Window Placement

When determining where to place a secondary window consider a number of factors, including the use of the window, the overall display dimensions, and the reason for the appearance of the window. In general, display a secondary window where it last appeared. If the user has not yet established a location for the window, place the window in a location that is convenient for the user to navigate to and that fully displays the window. If neither of these guidelines apply, horizontally center the secondary window within the primary window, just below the title bar, menu bar, and any docked toolbars.

## Modeless vs. Modal

A secondary window can be modeless or modal. A *modeless* secondary window allows the user to interact with either the secondary window or the primary window, just as the user can switch between primary windows. It is also well suited to situations where the user wants to repeat an action — for example, finding the occurrence of a word or formatting the properties of text.

A *modal* secondary window requires the user to complete interaction within the secondary window and close it before continuing with any further interaction outside the window. A secondary window can be modal in respect to its primary window or the system. In the latter case, the user must respond and close the window before interacting with any other windows or applications.

Because modal secondary windows restrict the user's choice, use them sparingly. Limit their use to situations when additional information is required to complete a command or when it is important to prevent any further interaction until satisfying a condition. Avoid using system modal secondary windows unless your application operates as a system level utility and then only use them in severe situations — for example, when an impending fatal system error or unrecoverable condition occurs.

## Default Buttons

When defining a secondary window, you can assign the ENTER key to activate a particular command button, called the *default button*, in the window. The system distinguishes the default button from other command buttons with a bold outline that appears around the button.

Define the default button to be the most likely action, such as a confirmation action or an action that applies transactions made in the secondary window. Avoid making a command button the default button if its action is irreversible or destructive. For example, in a text search and substitution window, do not use a Replace All button as the default button for the window.

You can change the default button as the user interacts with the window. For example, if the user navigates to a command button that is not the default button, the new button temporarily becomes the default. In such a case, the new default button takes on the default appearance, and the former default button loses the default appearance. Similarly, if the user moves the input focus to another control within the window that is not a command button, the original default button resumes being the default button.

The assignment of a default button is a common convention. However, when there is no appropriate button to designate as the default button or another control requires the ENTER key (for example, entering new lines in a multiline text control), you cannot define a default button for the window. In addition, when a particular control has the input focus and requires use of the ENTER key, you can temporarily have no button defined as the default. Then when the user moves the input focus out of the control, you can restore the default button.

Optionally, you can use double-clicking on single selection control, such as an option button or single selection list, as a shortcut technique to set or select the option and carry out the default button of the secondary window.

# Navigation in Secondary Windows

With the mouse and pen, navigation to a particular field or control involves the user pointing to the field and clicking or tapping it. For button controls, this action also activates that button. For example, for check boxes, it toggles the check box setting and for command buttons, it carries out the command associated with that button.

The keyboard interface for navigation in secondary windows uses the TAB and SHIFT+TAB keys to move between controls, to the next and previous control, respectively. Each control has a property that determines its place in the navigation order. Set this property such that the user can move through the secondary window following the usual conventions for reading: in western countries, left-to-right and top-to-bottom, with the primary control the user interacts with located in the upper left area of the window. Order controls such that the user can progress through the window in a logical sequence, proceeding through groups of related controls. Command buttons for handling overall window transactions are usually at the end of the order sequence.

You need not provide TAB key access to every control in the window. When using static text as a label, set the control you associated with it as the appropriate navigational destination, not the static text field itself. In addition, combination controls such as combo boxes, drop-down combo boxes, and spin boxes are considered single controls for navigational purposes. Because option buttons typically appear as a group, use the TAB key for moving the input focus to the current set choice in that group, but not between individual options — use arrow keys for this purpose. For a group of check boxes, provide TAB navigation to each control because their settings are independent of each other.

Optionally, you can also use arrow keys to support keyboard navigation between controls in addition to the TAB navigation technique wherever the interface does not require those keys. For example, you can use the UP ARROW and DOWN ARROW keys to navigate between single-line text boxes or within a group of check boxes or command buttons. Always use arrow keys to navigate between option button choices and within list box controls.

You can also use access keys to provide navigation to controls within a secondary window. This allows the user to access a control by pressing and holding the ALT key and an alphanumeric key that matches the access key character designated in the label of the control.

For more information about guidelines for selecting access keys, see Input Basics.

Unmodified alphanumeric keys also support navigation if the control that currently has the input focus does not use these keys for input. For example, if the input focus is currently on a check box control and the user presses an alphanumeric key, the input focus moves to the control with the matching access key. However, if the input focus is in a text box or list box, an alphanumeric key is used as text input for that control so the user cannot use it for navigation within the window without modifying it with the ALT key.

Access keys not only allow the user to navigate to the matching control, they have the same effect as clicking the control with the mouse. For example, pressing the access key for a command button carries out the action associated with that button. To ensure the user direct access to all controls, select unique access keys within a secondary window.

You can also use access keys to support navigation to a control, but then return the input focus to the control from which the user navigated. For example, when the user presses the access key for a specific command button that modifies the content of a list box, you can return the input focus to the list box after the command has been carried out.

OK and Cancel command buttons are typically not assigned access keys if they are the primary transaction keys for a secondary window. In this case, the ENTER and ESC keys, respectively, provide access to these buttons.

Pressing ENTER always navigates to the default command button, if one exists, and invokes the action associated with that button. If there is no current default command button, then a control can use the ENTER key for its own use.

## Validation of Input

Validate the user's input for a field or control in a secondary window as closely to the point of input as possible. Ideally, input is validated when it is entered for a particular field. You can either disallow the input, or use audio and visual feedback to alert the user that the data is not appropriate. You can also display a message box, particularly if the user repeatedly tries to enter invalid input. You can also reduce invalid feedback by using controls that limit selection to a specific set of choices — for example, check boxes, option buttons, drop-down lists — or preset the field with a reasonable default value.

If it is not possible to validate input at the point of entry, consider validating the input when the user navigates away from the control. If this is not feasible, then validate it when the transaction is committed, or whenever the user attempts to close the window. At that time, leave the window open and display a message; after the user dismisses the message, set the input focus to the control with the inappropriate data.

## Property Sheets and Inspectors

You can display the properties of an object in the interface in a number of ways. For example, some folder views display certain file system properties of an object. The image and name of an icon on the desktop also reflect specific properties of that object. You can also use other interface conventions, such as toolbars, status bars, or even scroll bars, to reflect certain properties. The most common presentation of an object's properties is a secondary window, called a property sheet. A *property sheet* is a modeless secondary window that displays the user-accessible properties of an object — that is, viewable, but not necessarily editable properties. Display a property sheet when the user chooses the Properties command for an object.

A *property inspector* is different from a property sheet — even when a property sheet window is modeless, the window is typically modal with respect to the object for which it displays properties. If the user selects another object, the property sheet continues to display the properties of the original object. A property inspector always reflects the current selection.

## Property Sheet Interface

The title bar text of the property sheet identifies the displayed object. If the object has a name, use its name and the word “Properties”. If the combination of the name plus “Properties” exceeds the width of the title bar, the system truncates the name and adds an ellipsis. If the object has no name, use the object’s type name. If the property sheet represents several objects, then also use the objects’ type name. Where the type name cannot be applied — for example, because the selection includes heterogeneous types — substitute the word “Selection” for the type name.

Because there can be numerous properties for an object and its context, you may need to categorize and group properties as sets within the property window. There are two techniques for supporting navigation to groups of properties in a property sheet. The first is a tabbed *property page*. Each set of properties is presented within the window as a page with a tab labeled with the name of the set. Use tabbed property pages for grouping peer-related property sets, as shown in Figure 8.2.

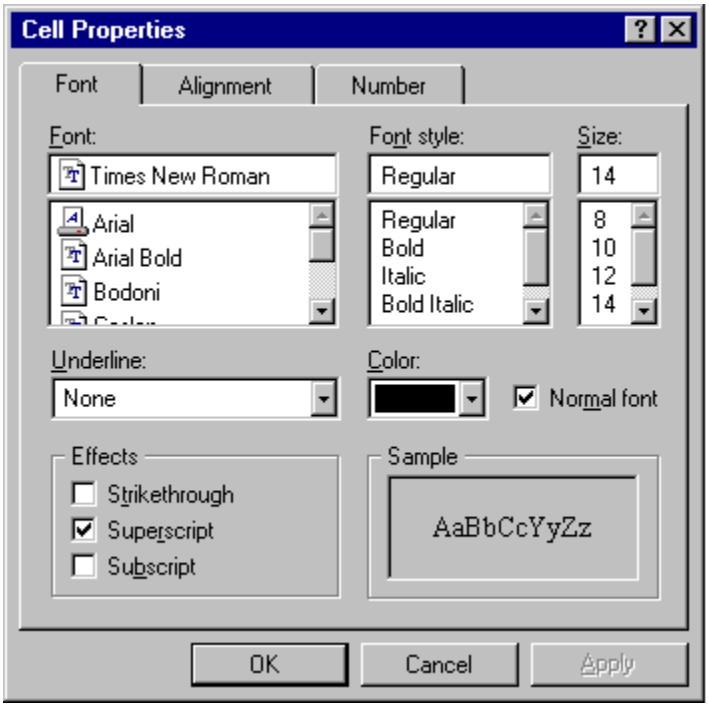



**Figure 8.2 A property sheet with tabbed pages**

When displaying the property sheet of an object, you can also provide access to the properties of the object’s immediate context or hierarchically related properties in the property sheet. For example, if the user selects text, you may want to provide access to the properties of the paragraph of that text in the same property sheet. Similarly, if the user selects a cell in a spreadsheet, you may want to provide access to its related row and column properties in the same property sheet. Although you can support this with additional tabbed pages, better access may be facilitated using another control — such as a drop-down list — to switch between groups of tabbed pages, as shown in Figure 8.3. This technique can also be used instead of multiple rows of tabs.

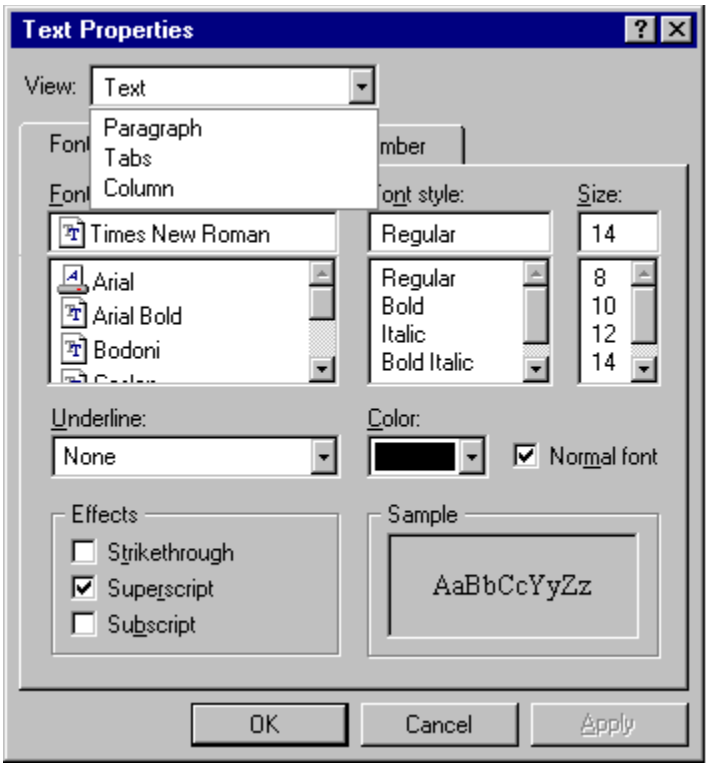



**Figure 8.3 A drop-down list for access to hierarchical property sets**

Where possible, make the values for properties found in property sheets transferable. You can support special transfer completion commands to enable copying only the properties of an object to another object. For example, you may want to support transferring data for text boxes or items in a list box.

For more details on transfer operations, see General Interaction Techniques.

## Property Sheet Commands

Property sheets typically allow the user to change the values for a property and then apply those transactions. Include the follow-ing common command buttons for handling the application of property changes.

| Command | Action |
|---|---|
| OK | Applies all pending changes and closes the property sheet window. |
| Apply | Applies all pending changes but leaves the property sheet window open. |
| Cancel | Discards any pending changes and closes the property sheet window. Does not cancel or undo changes that have already been applied. |

Optionally, you can also support a Reset command for canceling pending changes without closing the window.

You can also include other command buttons in property sheets. However, the location of command buttons within the property sheet window is very important. If you place a button on a property page, apply the action associated with the button to that page. For command buttons placed outside the page but still inside the window, apply the command to the entire window.

For the common property sheet transaction buttons — OK, Cancel, and Apply — it is best to place the buttons outside the pages because users consider the pages to be just a simple grouping or navigation technique. This means that if the user makes a change on one page, the change is not applied when the user switches pages. However, if the user makes a change on the new page and then chooses the OK or Apply command buttons, both changes are applied — or, in the case of Cancel, discarded.

If your design requires groups of properties to be applied on a page-by-page basis, then place OK, Cancel, and Apply command buttons on the property pages, always in the same location on each page. When the user switches pages, any property value changes for that page are applied, or you can prompt the user with a message box whether to apply or discard the changes.

You can include a sample in a property sheet window to illustrate a property value change that affects the object when the user applies the property sheet. Where possible, include the aspect of the object that will be affected in the sample. For example, if the user selects text and displays the property sheet for the text, include part of the text selection in the property sheets sample. If displaying the actual object — or a portion of it — in the sample is not practical, use an illustration that represents the object’s type.

## Closing a Property Sheet

If the user closes a property sheet window, follow the same convention as closing the content view of an object, such as a document. Avoid interpreting the Close button as Cancel. If there are pending changes that have not been committed, prompt the user to apply or discard the changes through a message box, as shown in Figure 8.4. If there are no unsaved changes, just close the window.

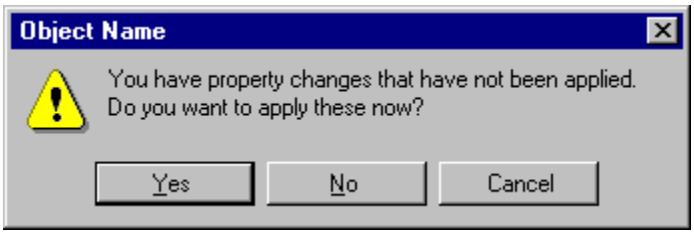



**Figure 8.4 Prompting for pending property changes**

If the user chooses the Yes button, the properties are applied and the message box window and the property sheet window are removed. If the user chooses the No button, the pending changes are discarded and the message box and property sheet windows are closed. Include a Cancel button in the message box, to allow the user to cancel the closing of the property sheet window.

## Property Inspectors

You can also display properties of an object using a dynamic viewer or browser that reflects the properties of the current selection. Such a property window is called a property inspector. When designing a property inspector, use a toolbar or palette window, or preferably a toolbar that the user can configure as a docked toolbar or palette window, as shown in Figure 8.5.

For more information about supporting docked and windowed toolbars, see Menus, Controls, and Toolbars. For more information about palette windows, see Palette Windows.


**Figure 8.5 A property inspector**

Apply property transactions that the user makes in a property inspector dynamically. That is, change the property value in the selected object as soon as the user makes the change in the control reflecting that property value.

Property inspectors and property sheets are not exclusive interfaces; you can include both. Each has its advantages. You can choose to display only the most common or frequently accessed properties in a property inspector and the complete set in the property sheet. You also can include multiple property inspectors, each optimized for managing certain types of objects.

As an option, you also can provide an interface for the user to change the behavior between a property sheet and a property inspector form of interaction. For example, you can provide a control on a property inspector that “locks” its view to be modal to the current object rather than tracking the selection.

## Properties of a Multiple Selection

When a user selects multiple objects and requests the properties for the selection, reflect the properties of all the objects in a single property sheet or property inspector rather than opening multiple windows. Where the property values differ, display the controls associated with those values using the mixed value appearance — sometimes referred to as the indeterminate state. However, also support the display of multiple property sheets when the user displays the property sheet of the objects individually. This convention provides the user with sufficient flexibility. If your design still requires access to individual properties when the user displays the property sheet of a multiple selection, include a control such as a list box or drop-down list in the property window for switching between the properties of the objects in the set.

## Properties of a Heterogeneous Selection

When a multiple selection includes different types of objects, include the intersection of the properties between the objects in the resulting property sheet. If the container of those selected objects treats the objects as if they were of a single type, the property sheet includes properties for that type only. For example, if the user selects text and an embedded object, such as a circle, and in that context an embedded object is treated as an element within the text stream, present only the text properties in the resulting property sheet.

## Properties of Grouped Items

When displaying properties, do not equate a multiple selection with a grouped set of objects. A group is a stronger relationship than a simple selection, because the aggregate resulting from the grouping can itself be considered an object, potentially with its own properties and operations. Therefore, if the user requests the properties of a grouped set of items, display the properties of the group or composite object. The properties of its individual members may or may not be included, depending on what is most appropriate.

## Dialog Boxes

A *dialog box* provides an exchange of information or dialog between the user and the application. Use a dialog box to obtain additional information from the user — information needed to carry out a particular command or task.

Because dialog boxes generally appear after choosing a particular menu item (including pop-up or cascading menu items) or a command button, define the title text for the dialog box window to be the name of the associated command. Do not include an ellipsis in the title text, even if the command menu name may have included one. Also, avoid including the command's menu title unless necessary to compose a reasonable title for the dialog box. For example, for a Print command on the File menu, define the dialog box window's title text as Print, not Print... or File Print. However, for an Object... command on an Insert menu, you can title the dialog box as Insert Object.

## Dialog Box Commands

Like property sheets, dialog boxes commonly include OK and Cancel command buttons. Use OK to apply the values in the dialog box and close the window. If the user chooses Cancel, the changes are ignored and the window is closed, canceling the operation the user chose. OK and Cancel buttons work best for dialog boxes that allow the user to set the parameters for a particular command. Typically, define OK to be the default command button when the dialog box window opens.

You can include other command buttons in a dialog box in addition to or replacing the OK and Cancel buttons. Label your command buttons to clearly define the button's purpose, but be as concise as possible. Long, wordy labels make it difficult for the user to easily scan and interpret a dialog box's purpose. Follow the design conventions for command buttons.

For more information about command buttons, see Menus, Controls, and Toolbars, and Visual Design.

## Layout

Orient controls in dialog boxes in the direction people read. In countries where roman alphabets are used, this means left to right, top to bottom. Locate the primary field with which the user interacts as close to the upper left corner as possible. Follow similar guidelines for orienting controls within a group in the dialog box.

Lay out the major command buttons either stacked along the upper right border of the dialog box or lined up across the bottom of the dialog box. Position the most important button — typically the default command — as the first button in the set. If you use the OK and Cancel buttons, group them together. You can use other arrangements if there is a compelling reason, such as a natural mapping relationship. For example, it makes sense to place buttons labeled North, South, East, and West in a compass-like layout. Similarly, a command button that modifies or provides direct support for another control may be grouped or placed next to those controls. However, avoid making that button the default button because the user will expect the default button to be in the conventional location.

## Common Dialog Box Interfaces

The system provides prebuilt interfaces for many common operations. Use these interfaces where appropriate. They can save you time while providing a high degree of consistency. If you customize or provide your own interfaces, maintain consistency with the basic functionality supported in these interfaces and the guidelines for their use. For example, if you provide your own property sheet for font properties, model your design to be similar in appearance and design to the common Font dialog box. Consistent visual and operational styles will allow users to more easily transfer their knowledge and skills.

The common dialog box interfaces have been revised from the ones provided in previous releases of Microsoft Windows.

## Open Dialog Box

The Open dialog box, as shown in Figure 8.6, allows the user to browse the file system, including direct browsing of the network, and includes controls to open a specified file. Use this dialog box to open files or browse for a filename, such as the File Open menu command or a Browse command button. Always set the title text to correctly reflect the command that displays the dialog box.

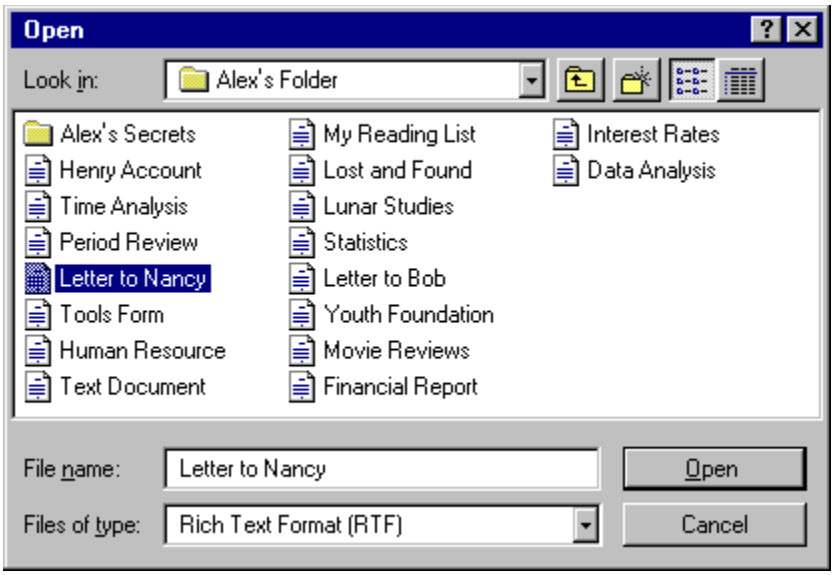



**Figure 8.6 The Open dialog box**

The system-supplied dialog box automatically handles the display of long filenames, direct manipulation transfers — such as drag and drop — and access to an icon's pop-up menus. The dialog box only displays filename extensions for files of registered types when the user selects this viewing option.

To open a file, the user selects a file from the list in the dialog box, or types a name in the File Name field and then chooses the Open command. The user can also display the pop-up menu for the file and choose its Open command. As a shortcut, double-clicking also opens the file. Choosing the Cancel button closes the window without opening the file.

When the user opens a shortcut icon, the dialog box opens the file of the object to which the link refers. In other words, the effect is the same as if the user directly opened the original file. Therefore, the name of the original file — not the name of the file link — should appear in the primary window's title bar.

The files listed in the dialog box reflect the current directory path and the type filter set in the Files Of Type drop-down list box. The list of files also includes shortcut icons in the current directory; these shortcut icons refer to file types that match the type filter.

The Look In drop-down list box displays the current directory. Displaying the list allows the user to view the hierarchy of the directory path and to navigate up the path tree. Tool buttons that are adjacent to this control provide the user with easy access to common functions. The dialog box also supports pop-up menus for the icons, the view in the list of files box, and the other controls in the window.

Set the default directory based on context. If the user opened the file directly, either from its location from the file system or using the Open dialog box, set the directory path to that location. If the user opened the application directly, then you can set the path as best fits the application. For example, an application may set up a default directory for its data files.

The user can change the directory path by selecting a different item in the Look In list, selecting a file

system container (such as a folder) in the list of files, or entering a valid path in the File Name field and choosing the Open button. Choosing the Cancel button should not change the path. Always preserve the latest directory path between subsequent openings of the dialog box. If the application supports opening multiple files, such as in MDI design, set the directory path to the last file opened, not the currently active child window. However, for multiple instances of an application, maintain the path separately for each instance.

Your application determines the default Files Of Type filter for the Open dialog box. This can be based on the last file opened, the last file type set by the user, or always a specific type, based on what most appropriately fits the context of the application.

The user can change the type filter by selecting a different type in the Files Of Type drop-down list box or by typing a filter into the File Name text box and choosing the Open button. Filters can include filename extensions. For example, if the user types in **.txt* and chooses the Open button, the list displays only files with the type extension of .TXT. Typing an extension into this text box also changes the respective type setting for the Files Of Type drop-down list box. If the application does not support that type, display the Files Of Type control with the mixed-case (indeterminate) appearance.

Include the types of files your application supports in the Files Of Type drop-down list box. For each item in the list, use a type description preferably based on the registered type names for the file types. For example, for text files, the type descriptor should be “Text Documents”. You can also include an “All Files” entry to display all files in the current directory, regardless of type.

When the user types a filename into the Open dialog box and chooses the Open button, the following conventions apply:

- The string includes no extension: the system attempts to use your application’s default extension or the current setting in the Files Of Type drop-down list box. For example, if the user types in *My Document*, and the application’s default extension is .DOC, then the system attempts to open My Document.doc. (The extension is not displayed.) If the user changes the type setting to Text Documents (*.txt), the file specification is interpreted as My Document.txt. If using the application’s default type or the type setting fails to find a matching file, the system attempts to open a file that appears in the list of files with the same name (regardless of extension). If more than one file matches, the first will be selected and the system displays a message box indicating multiple files match.
- The string includes an extension: the system first checks to see if it matches the application’s default type, any other registered types, or any extension in the Files Of Type drop-down list box. If it does not match, the system attempts to open it using the application’s default type or the current type setting in the Files Of Type drop-down list box. For example, Microsoft WordPad will open the file A Letter to Dr. Jones provided that: the file’s type matches the .DOC extension or the current type setting, and because the characters Jones (after the period) do not constitute a registered type. If this fails, the system follows the same behavior as for a file without an extension, checking for a match among the files that appear in the list of files.
- The string includes double-quotes at the beginning and end: the system interprets the string exactly, without the quotes and without appending any extension. For example, “My Document” is interpreted as My Document.
- The system fails to find a file: when the system cannot find a file, it displays a message box indicating that the file could not be found and advises the user to check the filename and path specified. However, your application may chose to handle this condition itself.
- The string the user types in includes invalid characters for a filename: the system displays a message box advising the user of this condition.

The Open dialog only handles the matching of a name to a file. It is your application’s responsibility to ensure the format of the file is valid, and if not, to appropriately notify the user.

## Save As Dialog Box

The Save As dialog box, as shown in Figure 8.7, is designed to save a file using a particular name, location, type, and format. Typically, applications that support the creation of multiple user files provide this command. However, if your application maintains only private data files and automatically updates those files, this dialog box may not be appropriate.

Display this dialog box when the user chooses the Save As command or file-oriented commands with a similar function, such as the Export File command. Also display the Save As dialog box when the user chooses the Save command, and has not supplied or confirmed a filename. If you use this dialog box for other tasks that require saving files, define the title text of the dialog box to appropriately reflect that command.

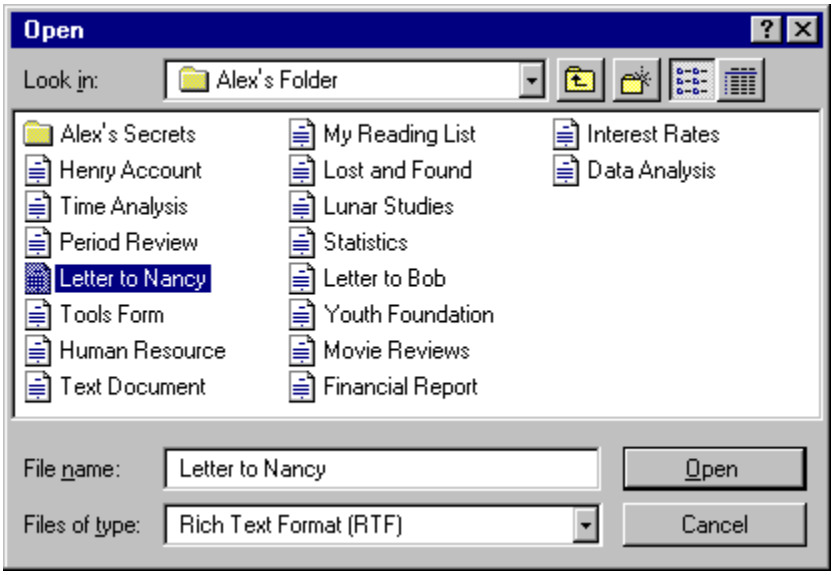



**Figure 8.7 The Save As dialog box**

The appearance and operation of the Save As dialog box is similar to the Open dialog box, except that the type field — the Save As Type drop-down list box — defines the default type for the saved file; it also filters the list of files displayed in the window.

To save a file, the user chooses the Save button and saves the file with the name that appears in the File Name text box. Although the user can type in a name or select a file from the list of files, your application should preset the field to the current name of the file. If the file has not been named yet, propose a name based on the registered type name for the file — for example, Text Document (2).

For more information about naming files, see Windows, and Integrating with the System.

The Save In drop-down list box indicates the immediate container in the directory path (or folder). The user can change the path using this control and the list of files box. If the file already exists, always save the file to its original location. This means that the current path for the Save As dialog box should always be set to the path where the file was last saved. If the file has never been saved, save the file with your application's default path setting or to the location defined by the user, either by typing in the path or by using the controls in the dialog box.

If the user chooses the Cancel button in the Save As dialog box, do not save the file or other settings. Restore the path to its original setting.

Include the file types supported by your application in the Save As Type drop-down list box. You may

need to include a format description as part of a type name description. Although a file's format can be related to its type, a format and a type are not the same thing. For example, a bitmap file can be stored in monochrome, 16 , 256 or 24-bit color format, but the file's type is the same for all of them. Consider using the following convention for the items you include as type descriptions in the Save As Type drop-down list box.

*Type Name* [*Format Description*]

When the user supplies a name of the file, the Save As dialog box follows conventions similar to the Open dialog box. If the user does not include an extension, the system uses the setting in the Save As Type drop-down list or your application's default file type. If the user includes an extension, the system checks to see if the extension matches your application's default extension or a registered extension. If it does, the system saves the file as the type matching that extension. (The extension is hidden unless the system is set to display extensions.) Otherwise, the system interprets the user-supplied extension as part of the filename and appends the extension set in the Save As Type field. Note that this only means that the type (extension) is set. The format may not be correct for that type. It is your application's responsibility to write out the correct format.

Make certain you preserve the creation date for files that the user opens and saves. If your application saves files by creating a temporary file then deletes the original, renaming the temporary file to the original filename, be certain you copy the creation date from the original file. Certain system file management functionality may depend on preserving the identity of the original file.

If the user types in a filename beginning and ending with double quotes, the system saves the file without appending any extension. If the string includes a registered extension, the file appears as that type. If the user supplies a filename with invalid characters or the specified path does not exist, the system displays a message box, unless your application handles these conditions.

Here are some examples of how the system saves user supplied filenames. Examples assume .TXT as the application's default type or the Save As Type setting.

| **What the user types** | **How system saves the file** | **Description** |
|---|---|---|
| My File | My File.txt | Type is based on the file type established in Save As Type drop-down list box or the application's default type. |
| My File.txt | My File.txt | Type must match the application's default type or a registered type. |
| My File for Mr. Jones | My File for Mr. Jones.txt | . Jones does not qualify as a registered type or a type included in the Save As Type drop-down list box, so the type is appended based on the Save As Type setting or the application's default type. |
| My File for Mr. Jones.txt | My File for Mr. Jones.txt | Type must match a registered type or a type included in the Save As Type drop-down list box. |
| "My File" | My File | Type will be unknown. The file is saved exactly |

| | | |
|---|---|---|
| | | as the string between the quotes appears. |
| “My File.txt” | My File.txt | No type is appended. The file is saved exactly as the string between the quotes appears. |
| My File. | My File..txt | Type is based on the Save As Type drop-down list box or the application’s default type. |
| “My File.” | My File. | Type will be unknown. |
| “My” File | File is not saved. | System (or application) displays a message box notifying the user of invalid filename. |

## Find and Replace Dialog Boxes

The Find and Replace dialog boxes provide controls that search for a text string specified by the user and optionally replace it with a second text string specified by the user. These dialog boxes are shown in Figure 8.8.

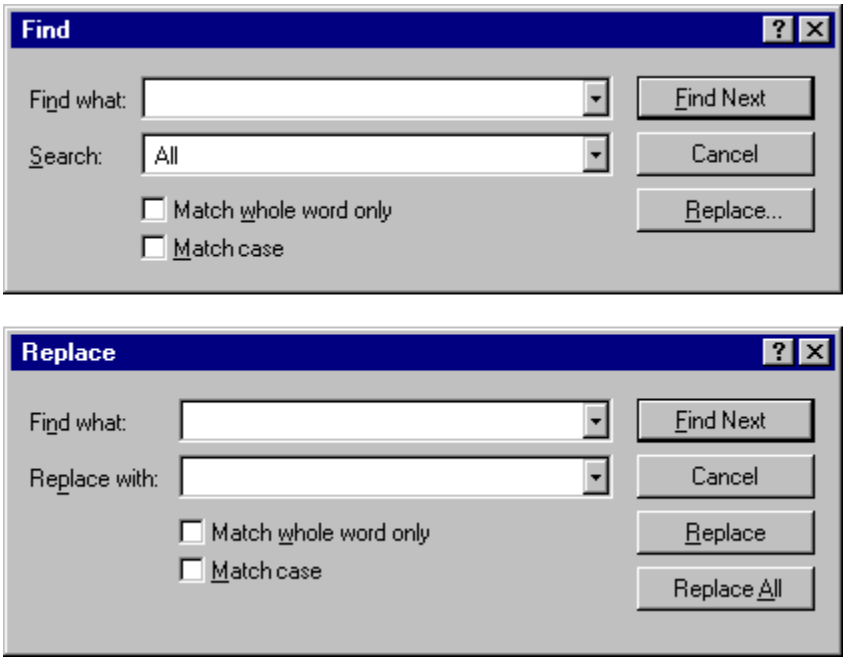



**Figure 8.8 The Find and Replace dialog boxes**

## Print Dialog Box

The Print dialog box, shown in Figure 8.9, allows the user to select what to print, the number of copies to print, and the collation sequence for printing. It also allows the user to choose a printer and provides a command button that provides shortcut access to that printer's properties.

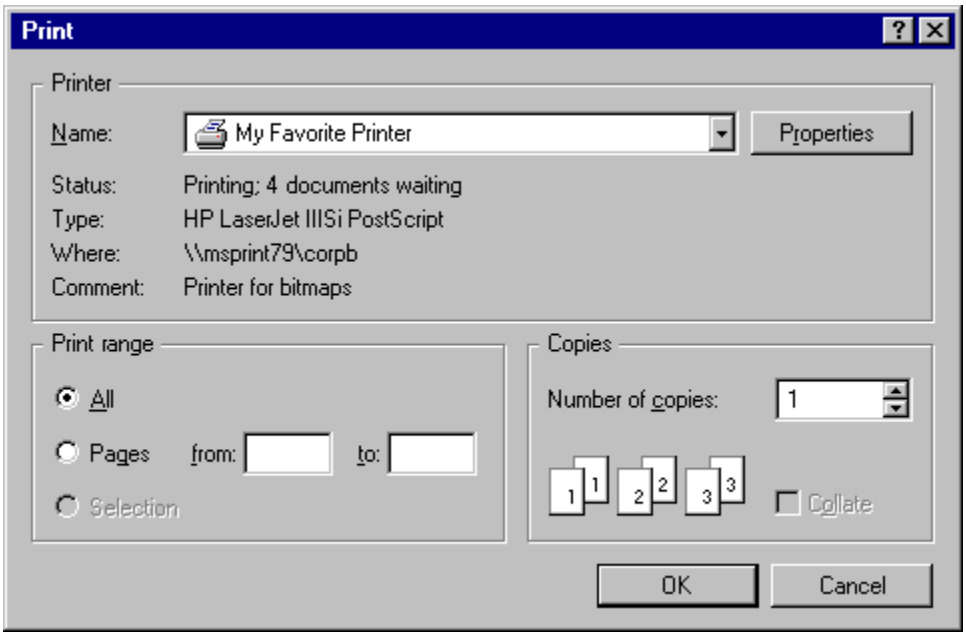



**Figure 8.9 The Print dialog box**

# Print Setup Dialog Box

The Print Setup dialog box displays the list of available printers and provides controls for selecting a printer and setting paper orientation, size, source, and other printer properties.

Do not include this dialog box if you are creating or updating your application for Microsoft Windows 95 or later releases.

## Page Setup Dialog Box

The Page Setup dialog box, as shown in Figure 8.10, provides controls for specifying properties about the page elements and layout.

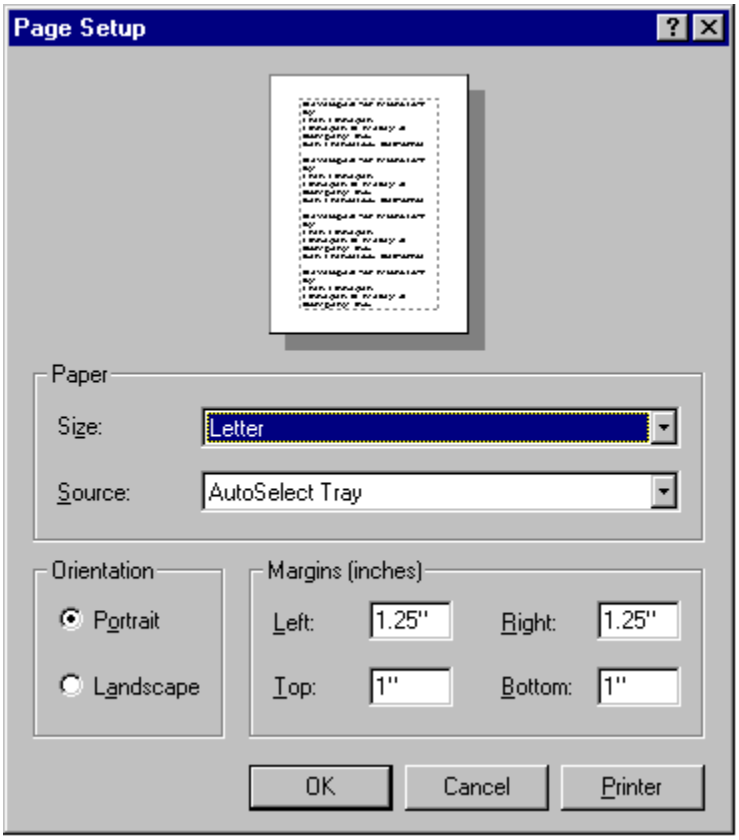



**Figure 8.10 Page Setup interface used as a dialog box**

In this context, page orientation refers to the orientation of the page and not the printer, which may also have these properties. Generally, the page's properties override those set by the printer, but only for the printing of that page or document.

The Printer button in the dialog box displays a supplemental dialog box (as shown in Figure 8.11) that provides information on the current default printer. Similarly to the Print dialog box, it displays the current property settings for the default printer and a button for access to the printer's property sheet.

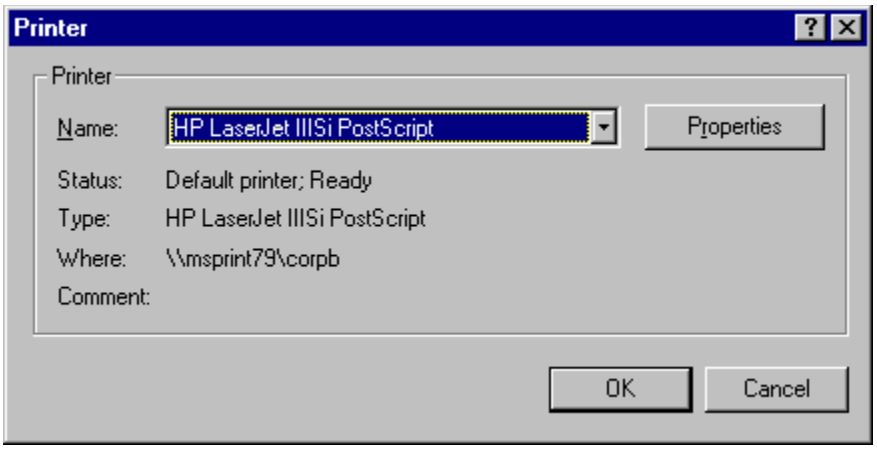



**Figure 8.11 The supplemental Printer dialog box**

## Font Dialog Box

This dialog box displays the available fonts and point sizes of the available fonts installed in the system. Your application can filter this list to show only the fonts applicable to your application. You can use the Font dialog box to display or set the font properties of a selection of text. Figure 8.12 shows the Font dialog box.

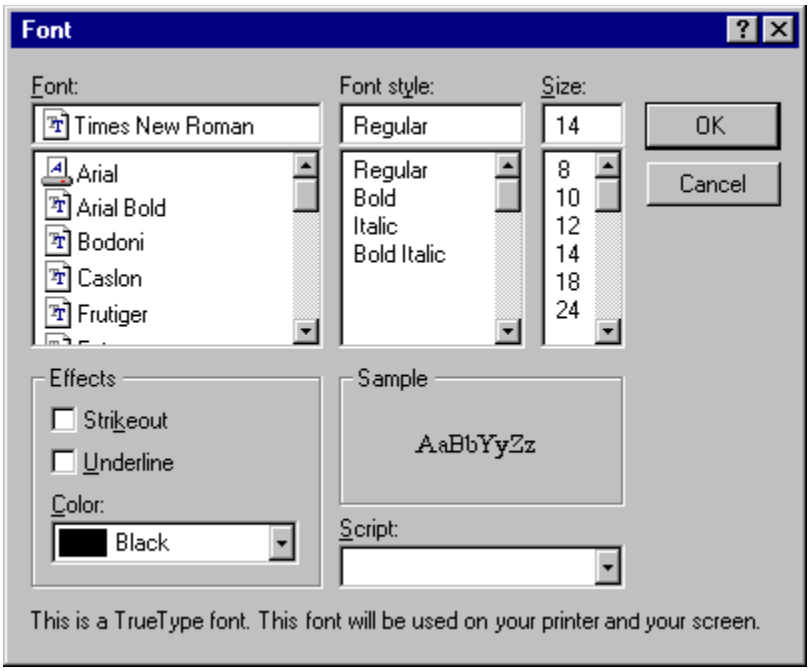



**Figure 8.12 The Font dialog box**

## Color Dialog Box

The Color dialog box (as shown in Figure 8.13) displays the available colors and includes controls that allow the user to define custom colors. You can use this control to provide an interface for users to select colors for an object.


**Figure 8.13 The Color dialog box (unexpanded appearance)**

The Basic Colors control displays a default set of colors. The number of colors displayed here is determined by the installed display driver. The Custom Colors control allows the user to define more colors using the various color selection controls provided in the window.

Initially, you can display the dialog box as a smaller window with only the Basic Colors and Custom Colors controls and allow the user to expand the dialog box to define additional colors (as shown in Figure 8.14).

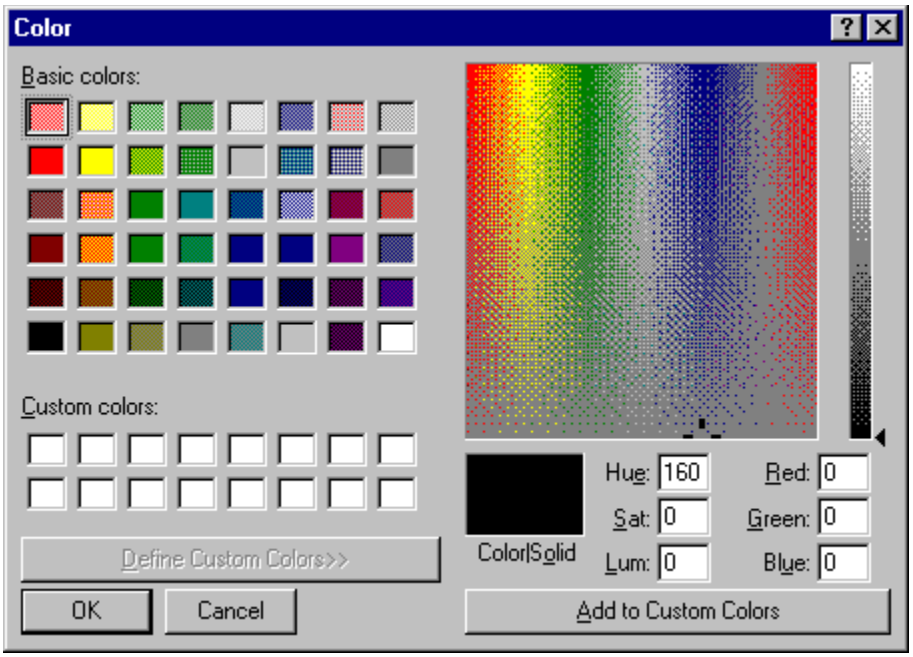



**Figure 8.14 The Color dialog box (expanded)**

## Palette Windows

Palette windows are modeless secondary windows that present a set of controls. For example, when toolbar controls appear as a window, they appear in a palette window. Palette windows are distinguished by their visual appearance. The height of the title bar for a palette window is shorter, but it still includes only a Close button in the title area, as shown in 8.15.

For more information about toolbars and palette windows, see Menus, Controls, and Toolbars.


**Figure 8.15 A palette window**

Make the title text for a palette window the name of the command that displays the window or the name of the toolbar it represents. The system supplies default size and font settings for the title bar and title bar text for palette windows.

The title bar height and font size settings can be accessed using the **SystemParametersInfo** function. For more information about this function, see the documentation included in the Microsoft Win32 Software Development Kit (SDK).

You can define palette windows as a fixed size, or, more typically, sizable by the user. Two visual cues indicate when the window is sizable: changing the pointer image to the size pointer, and placing a Size command in the window's pop-up menu. Preserve the window's size and position so the window can be restored if it, or its associated primary window, is closed.

Like other windows, the title bar and the border areas provide an access point for the window's pop-up menu. Commands on a palette window's pop-up menu can include Close, Move, Size (if sizable), Always On Top, and Properties, as shown in Figure 8.16.

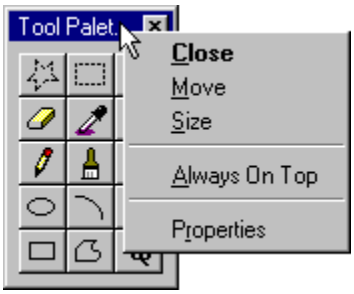



**Figure 8.16 A pop-up menu for a palette window**

Including the Always On Top command or property in the window's property sheet allows the user to configure the palette window to always stay at the top of the Z order of the window set of which it is a part. Turning off this option keeps the palette window within its set of related windows, but allows the user to have other windows of the set appear on top of the palette window. This feature allows the user to configure preferred access to the palette window.

You can also include a Properties command on the palette window’s pop-up menu to provide an interface for allowing the user to edit properties of the window, such as the Always On Top property, or a means of customizing the content of the palette window.

## Message Boxes

A message box is a secondary window that displays a message; information about a particular situation or condition. Messages are an important part of the interface for any software product. Messages that are too generic or poorly written frustrate users, increase support costs, and ultimately reflect on the quality of the product. Therefore, it is worthwhile to design effective message boxes.

However, it is even better to avoid creating situations that require you to display a message. For example, if there may be insufficient disk space to perform an operation, rather than assuming that you will display a message box, check before the user attempts the operation and disable the command.

## Title Bar Text

Use the title bar of a message box to appropriately identify the context in which the message is displayed — usually the name of the object. For example, if the message results from editing a document, the title text is the name of that document, optionally followed by the application name. If the message results from a nondocument object, then use the application name. Providing an appropriate identifier for the message is particularly important in the Windows multitasking environment, because message boxes might not always be the result of current user interaction. In addition, because OLE technology allows objects to be embedded, different application code may be running when the user activates the object for visual editing. Therefore, the title bar text of a message box provides an important role in communicating the source of a message. Do not use descriptive text for message box title text such as “warning” or “caution.” The message symbol conveys the nature of the message. Never use the word “error” in the title text.

## Message Box Types

Message boxes typically include a graphical symbol that indicates what kind of message is being presented. Most messages can be classified in one of the categories shown in Table 8.1.

**Table 8.1 Message Types and Associated Symbols**

| Symbol | Message type | Description | |
|---|---|---|---|
|  | Information | Provides information about the results of a command. Offers no user choices; the user acknowledges the message by clicking the OK button. |   |
|  | Warning | Alerts the user to a condition or situation that requires the user's decision and input before proceeding, such as an impending action with potentially destructive, irreversible consequences. The message can be in the form of a question — for example, "Save changes to MyReport?". |   |
|  | Critical | Informs the user of a serious problem that requires intervention or correction before work can continue. |   |

The system also includes a question mark message symbol. This message symbol (as shown in Figure 8.17) was used in earlier versions of Windows for cautionary messages that were phrased as a question.

**Figure 8. 17 Inappropriate message symbol**

However, the message icon is no longer recommended as it does not clearly represent a type of message and the phrasing of a message as a question could apply to any message type. In addition, users can confuse the message symbol question mark with Help information. Therefore, do not use this question mark message symbol in your message boxes. The system continues to support its inclusion only for backward compatibility.

You can include your own graphics or animation in message boxes. However, limit your use of these types of message boxes and avoid defining new graphics to replace the symbols for the existing standard types.

Because a message box disrupts the user's current task, it is best to display a message box only when the window of the application displaying the message box is active. If it is not active, then the

application uses its entry in the taskbar to alert the user. Once the user activates the application, the message box can be displayed. Display only one message box for a specific condition. Displaying a sequential set of message boxes tends to confuse users.

For more information about how to use the taskbar to notify the user when the application may not be active, see Integrating with the System.

You can also use message boxes to provide information or status without requiring direct user interaction to dismiss them. For example, message boxes that provide a visual representation of the progress of a particular process automatically disappear when the process is complete, as shown in Figure 8.18. Similarly, product start-up windows that identify the product name and copyright information when the application starts can be automatically removed once the application has loaded. In these situations, you do not need to include a message symbol. Use this technique only for noncritical, informational messages, as some users may not be able to read the message within the short time it is displayed.

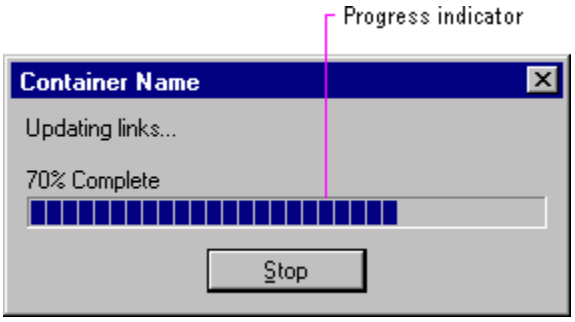



**Figure 8.18 A progress message box**

## Command Buttons in Message Boxes

Typically, message boxes contain only command buttons as the appropriate responses or choices offered to the user. Designate the most frequent or least destructive option as the default command button. Command buttons allow the message box interaction to be simple and efficient. If you need to add other types of controls, always consider the potential increase in complexity.

If a message requires no choices to be made but only acknowledgment, use an OK button — and, optionally, a Help button. If the message requires the user to make a choice, include a command button for each option. The clearest way to present the choices is to state the message in the form of a question and provide a button for each response. When possible, phrase the question to permit Yes or No answers, represented by Yes and No command buttons. If these choices are too ambiguous, label the command buttons with the names of specific actions — for example, “Save” and “Delete.”

You can include command buttons in a message box that correct the action that caused the message box to be displayed. For example, if the message box indicates that the user must switch to another application window to take corrective action, you can include a button that switches the user to that application window. Be sure, however, to make the result of any such button’s action very clear.

Some situations may require offering the user not only a choice between performing or not performing an action, but an opportunity to cancel the process altogether. In such situations, use a Cancel button, as shown in Figure 8.19. Be sure, however, to make the result of any such button’s action very clear.

When using Cancel as a command button in a message box, remember that to users, Cancel implies restoring the state of the process or task that started the message. If you use Cancel to interrupt a process and the state cannot be restored, use Stop instead.

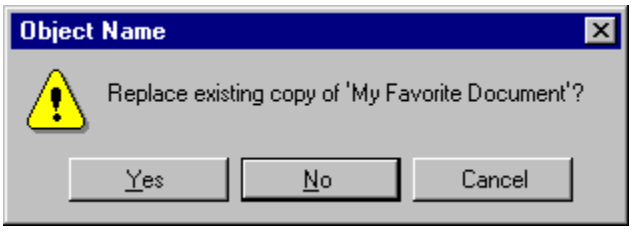



**Figure 8.19 Message box choices**

## Message Box Text

The message text you include in a message box should be clear, concise, and in terms that the user understands. This usually means using no technical jargon or system-oriented information.

In addition, observe the following guidelines for your message text:

- State the problem, its probable cause (if possible), and what the user can do about it — no matter how obvious the solution may seem to be. For example, instead of “Insufficient disk space,” use “‘Sample Document’ could not be saved, because the disk is full. Try saving to another disk or freeing up space on this disk.”
- Consider making the solution an option offered in the message. For example, instead of “One or more of your lines are too long. The text can only be a maximum of 60 characters wide,” you might say, “One or more of your lines are too long. Text can be a maximum of 60 characters in Portrait mode or 90 characters wide in Landscape. Do you want to switch to Landscape mode now?” Offer Yes and No as the choices.
- Avoid using unnecessary technical terminology and overly complex sentences. For example, “picture” can be understood in context, whereas “picture metafile” is a rather technical concept.
- Avoid phrasing that blames the user or implies user error. For example, use “Cannot find filename” instead of “Filename error.” Avoid the word “error” altogether.
- Make messages as specific as possible. Avoid mapping more than two or three conditions to a single message. For example, there may be several reasons why a file cannot be opened; provide a specific message for each condition.
- Avoid relying on default system-supplied messages, such as MS-DOS® extended error messages and Kernel INT 24 messages; instead, supply your own specific messages wherever possible.
- Be brief, but complete. Provide only as much background information as necessary. A good rule of thumb is to limit the message to two or three lines. If further explanation is necessary, provide this through a command button that opens a Help window.

You may also include a message identification number as part of the message text for each message for support purposes. However, to avoid interrupting the user’s ability to quickly read a message, place such a designation at the end of the message text and not in the title bar text.

## Pop-up Windows

Use pop-up windows to display additional information when an abbreviated form of the information is the main presentation. For example, you could use a pop-up window to display the full path for a field or control, when an entire path cannot be presented and must be abbreviated. Pop-up windows are also used to provide context-sensitive Help information, as shown in Figure 8.20.

For more information about using pop-up windows for Help information, see User Assistance.

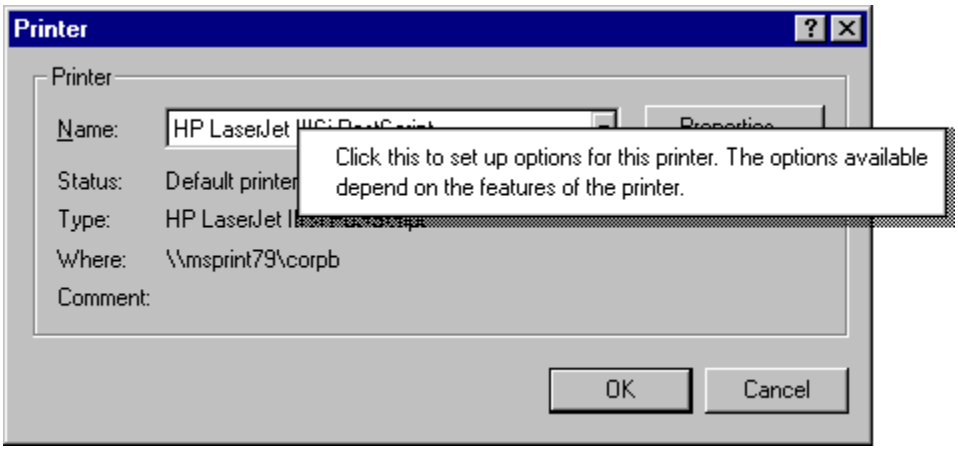



**Figure 8.20 A context-sensitive Help pop-up window**

Tooltips are another example of a pop-up window used to display contextual information, by providing the names for controls in toolbars. The writing tool is also another example of the use of a pop-up window.

How pop-up windows are displayed depends on their use, but the typical means is by the user either pointing or clicking with mouse button 1 (for pens, tapping), or an explicit command. If you use pointing as the technique to display a pop-up window, display the window after a time-out. The system automatically handles time-outs if you use the standard tooltip controls. If you are providing your own implementation, you can use the current double-click speed setting as a metric for displaying and removing the pop-up window.

If you use clicking to display a pop-up window, change the pointer as feedback to the user indicating that the pop-up window exists and requires a click. From the keyboard, you can use the Select key (SPACEBAR) to open and close the window.

# Introduction

User tasks can often involve working with different types of information, contained in more than one window or view. There are different techniques that you can use to manage a set of windows or views. This topic covers some common techniques and the factors to consider for selecting a particular model.

## Single Document Window Interface

In many cases, the interface of an object or application can be expressed using a single primary window with a set of supplemental secondary windows. The desktop and taskbar provide management of primary windows. Opening a window puts it at the top of the Z order and places an entry on the taskbar, making it easier for users to switch between windows without having to shuffle or reposition them.

By supporting a single instance model where you activate an existing window (within the same desktop) if the user reopens the object, you make single primary windows more manageable, and reduce the potential confusion for the user. This also provides a data-centered, one-to-one relationship between an object and its window.

In addition, Microsoft OLE supports the creation of compound documents or other types of information containers. Using these constructs, the user can assemble a set of different types of objects for a specific purpose within a single primary window, eliminating the necessity of displaying or editing information in separate windows.

For more information about OLE, see Working with OLE Embedded and OLE Linked Objects.

Some types of objects, such as device objects, may not even require a primary window and use only a secondary window for viewing and editing their properties. When this occurs, do not include the Open command in the menu for the object; instead, replace it with a Properties command, defined as the object's default command.

It is also possible for an object to have no windows; an icon is its sole representation. In this very rare case, make certain that you provide an adequate set of menu commands to allow a user to control its activity.

## Multiple Document Interface

For some tasks, the taskbar may not be sufficient for managing a set of related windows; for example, it can be more effective to present multiple views of the same data or multiple views of related data in windows that share interface elements. You can use *multiple document interface* (MDI) for this kind of situation.

The MDI technique uses a single primary window, called a *parent window*, to visually contain a set of related *document* or *child windows*, as shown in Figure 9.1. Each child window is essentially a primary window, but is constrained to appear only within the parent window instead of on the desktop. The parent window also provides a visual and operational framework for its child windows. For example, child windows typically share the menu bar of the parent window and can also share other parts of the parent's interface, such as a toolbar or status bar. You can change these to reflect the commands and attributes of the active child window.

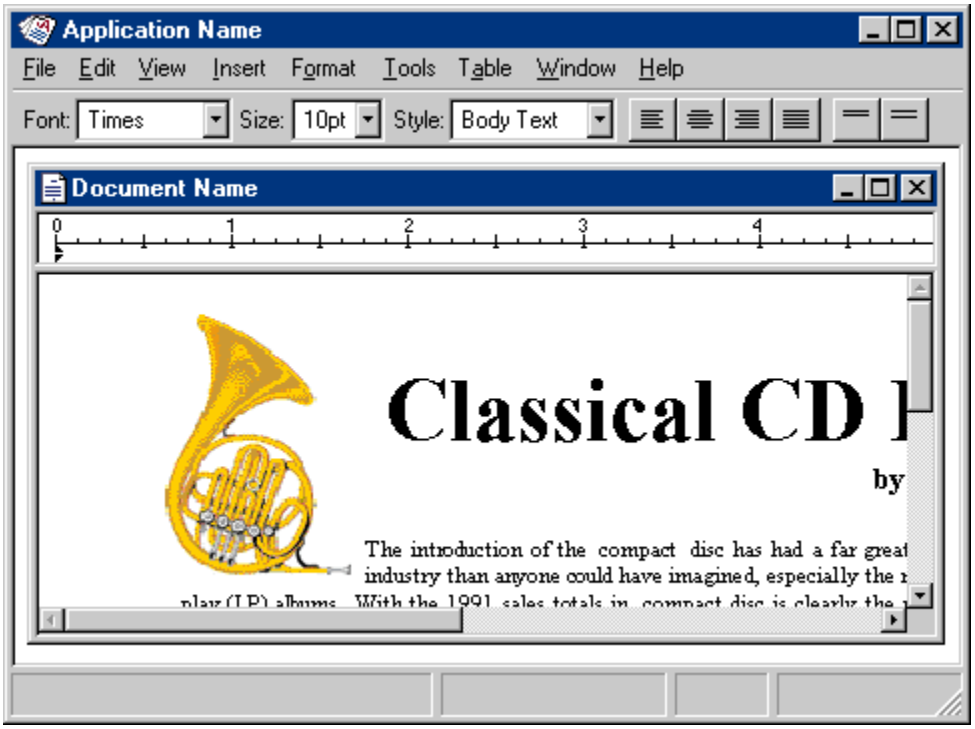



**Figure 9.1 An MDI parent and child window**

Secondary windows — such as dialog boxes, message boxes, or property sheets — displayed as a result of interaction within the MDI parent or child, are typically not contained or clipped by the parent window. These windows should be activated and displayed following the common conventions for secondary windows associated with a primary window, even if they apply to individual child windows.

For more information about the interaction between a primary window and its secondary windows, see Windows, and Secondary Windows.

For the title bar of an MDI parent window, include the icon and name of the application or the object that represents the work area displayed in the parent window. For the title bar of a child window, include the icon representing the document or data file type and its filename. Also support pop-up menus for the window and the title bar icon for both the parent window and any child windows.

## Opening and Closing MDI Windows

The user starts an MDI application either by directly opening the application or by opening a document (or data file) of the type supported by the MDI application. If directly opening an MDI document, the MDI parent window opens first and then the child window for the file opens within it. To support the user opening other documents associated with the application, include an interface, such as an Open dialog box.

When the user directly opens an MDI document outside the interface of its MDI parent window — for example, by double-clicking the file — if the parent window for the application is already open, open another instance of the MDI parent window rather than the document's window in the existing MDI parent window. Although the opening of the child window within the existing parent window can be more efficient, the opening of the new window can disrupt the task environment already set up in that parent window. For example, if the newly opened file is a macro, opening it in the opened parent window could inadvertently affect other documents open in that window. If the user wishes to open a file as part of the set in a particular parent MDI window, the commands within that window provide that support.

Because MDI child windows are primary windows, support closing them following the same conventions for primary windows by including a Close button in their title bars and a Close command in their pop-up menu for the windows. When the user closes a child window, any unsaved changes are processed following these common conventions for primary windows. Do not close its parent window, unless the parent window does not provide context or operations without an open child window.

When the user closes the parent window, close all of its child windows. Where possible, preserve the state of a child window, such as its size and position within the parent window; restore the state when the user reopens the file.

## Moving and Sizing MDI Windows

MDI allows the user to move or hide the child windows as a set by moving or minimizing the parent window. When the user moves an MDI parent window, maintain the relative positions of the open child windows within the parent window. Moving a child window constrains it to its parent window; in some cases, the size of the parent window's interior area may result in clipping a child window. Optionally, you can support automatic resizing of the parent window when the user moves or resizes a child window either toward or away from the edge of the parent window.

The recommended visual appearance of a minimized child window in Microsoft Windows is now that of a window that has been sized down to display only part of its title area and its border. This avoids potential confusion between minimized child window icons and icons that represent objects.

Although an MDI parent window minimizes as an entry on the taskbar, MDI child windows minimize within their parent window, as shown in Figure 9.2.

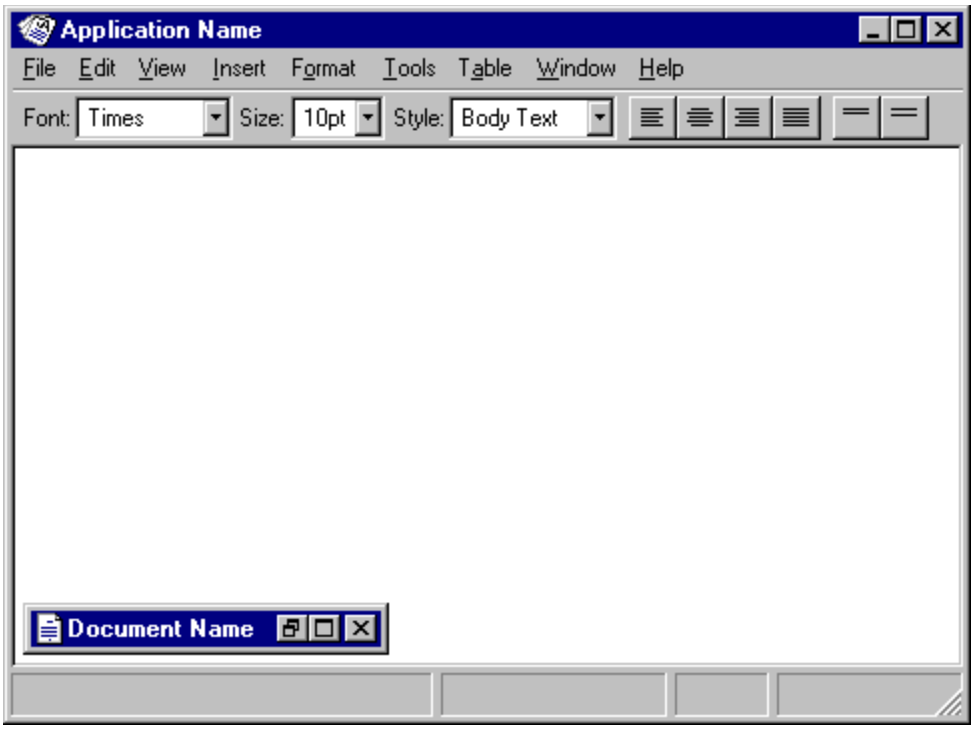



**Figure 9.2 A minimized MDI child window**

When the user maximizes an MDI parent window, expand the window to its maximum size, like any other primary window. When the user maximizes an MDI child window, also expand it to its maximum size. When this size exceeds the interior of its parent window, merge the child window with its parent window. The child window's title bar icon, Restore button, Close button, and Minimize button (if supported) are placed in the menu bar of the parent window in the same relative position as in the title bar of the child window, as shown in Figure 9.3. Append the child window title text to the parent window's title text.

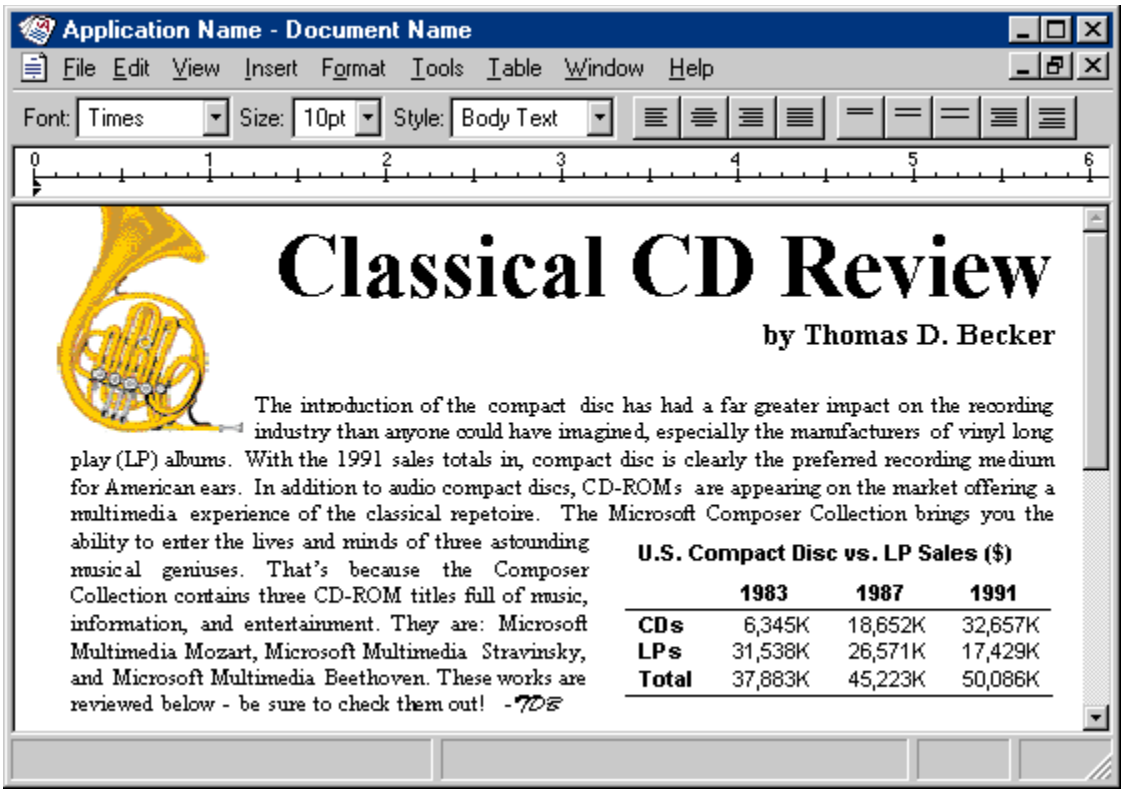



**Figure 9.3 A maximized MDI child window**

If the user maximizes one child window and it merges with the parent window and then switches to another, display that window as maximized. Similarly, when the user restores one child window from its maximized state, restore all other child windows to their previous sizes.

## Switching Between MDI Child Windows

Apply the same common mouse conventions for activating and switching between primary windows for MDI child windows. CTRL+F6 and CTRL+TAB (and SHIFT+ modified combinations to cycle backwards) are the recommended keyboard shortcuts for switching between child windows. In addition, include a Window menu on the menu bar of the parent window with commands for switching between child windows and managing or arranging the windows within the MDI parent window — such as Tile or Cascade.

When the user switches child windows, you can change the interface of the parent window — such as its menu bar, toolbar, or status bar — to appropriately reflect the commands that apply to that child window. However, provide as much consistency as possible, keeping constant any menus that represent the document files and control the application or overall parent window environment, such as the File menu or the Window menu.

## MDI Alternatives

MDI does have its limitations. MDI reinforces the visibility of the application as the primary focus for the user. Although the user can start an MDI application by directly opening one of its document or data files, to work with multiple documents within the same MDI parent window, the user uses the application's interface for opening those documents.

When the user opens multiple files within the same MDI parent window, the storage relationship between the child windows and the objects being viewed in those windows is not consistent. That is, although the parent window provides visual containment for a set of child windows, it does not provide containment for the files those windows represent. This makes the relationship between the files and their windows more abstract, making MDI more challenging for beginning users to learn.

Similarly, because the MDI parent window does not actually contain the objects opened within it, MDI cannot support an effective design for persistence. When the user closes the parent window and then reopens it, the context cannot be restored because the application state must be maintained independently from that of the files last opened in it.

MDI can make some aspects of the OLE interface unintentionally more complex. For example, if the user opens a text document in an MDI application and then opens a worksheet embedded in that text document, the task relationship and window management breaks down, because the embedded worksheet's window does not appear in the same MDI parent window.

Finally, the MDI technique of managing windows by confining child windows to the parent window can be inconvenient or inappropriate for some tasks, such as designing with window or form layout tools. Similarly, the nested nature of child windows may make it difficult for the user to differentiate between a child window in a parent window versus a primary window that is a peer with the parent window, but positioned on top.

Although MDI provides useful conventions for managing a set of related windows, it is not the only means of supporting task management. Some of its window management techniques can be applied in some alternative designs. The following — workspaces, workbooks, and projects — are examples of some possible design alternatives. They present a single window design model, but in such a way that preserves some of the window and task management benefits found in MDI.

Although these examples suggest a form of containment of multiple objects, you can also apply some of these designs to display multiple views of the same data. Similarly, these alternatives may provide greater flexibility with respect to the types of objects that they contain. However, as with any container, you can define your implementation to hold and manage only certain types of objects. For example, an appointment book and an index card file are both containers that organize a set of information but may differ in the way they display that information and the type of information they manage. Whether you define a container to hold the same or different types of objects depends on the design and purpose of the container.

The following examples illustrate alternatives of data-centered window or task management. They are not exclusive of other possible designs. They are intended only as suggestive possibilities, rather than standard constructs. As a result, the system does not include these constructs and provides no explicit programming interfaces. In addition, some specific details are left to you to define.

## Workspaces

A *workspace* shares many of the characteristics of MDI, including the association and management of a set of related windows within a parent window, and the sharing of the parent window's interface elements, such as menus, toolbars, and status bar. Figure 9.4 shows an example of a workspace.

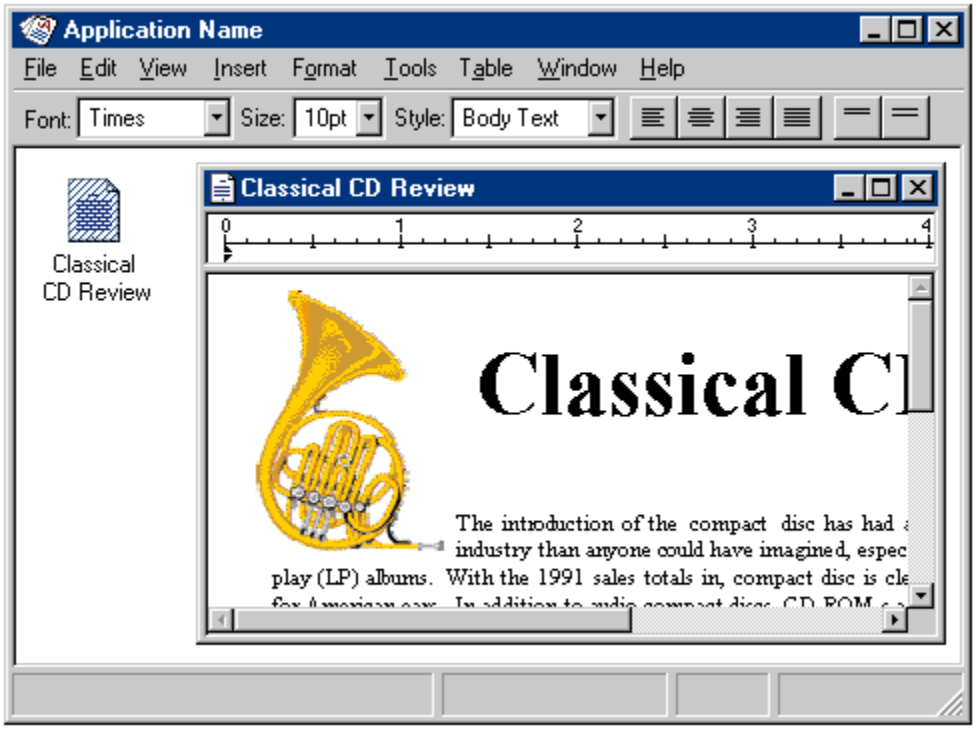



**Figure 9.4 Example of a workspace design**

## Workspaces as a Container

Based on the metaphor of a work area, like a table, desktop, or office, a workspace differs from an MDI by including the concept of containment. Objects contained or stored in the workspace can be presented in the same way files appear in folders. However, objects within a workspace open as child windows within the workspace parent window. In this way, a workspace's behavior is similar to that of the desktop, except that a workspace itself is an object that can be displayed as an icon and opened into a window. To have an object's window appear in the workspace, the object must reside there.

The actual storage mechanism you use depends on the type of container you implement. The content of the parent window can represent a single file, or you can devise your own mechanism to map the content into the file system. Consider using OLE in your implementation to facilitate interaction between your workspace, the shell, and other applications. For example, you may want to support the user moving objects from the workspace into other containers, such as the desktop and folders. However, if you do, when the user opens the object, it should appear in its own window, not the workspace window — with its interface elements, such as a menu bar — also appearing within its own window.

The workspace is an object itself and therefore you should define its specific commands and properties. You can also include commands for creating new objects within the workspace and, optionally, a Save All command that saves the state of all the objects opened in the workspace.

## Workspaces for Task Grouping

Because a workspace visually contains and constrains the icons and windows of the objects placed in it, you can define workspaces to allow the user to organize a set of objects for particular tasks. Like MDI, this makes it easy for the user to move or switch to a set of related windows as a set.

Also similar to MDI, the child windows of objects opened in the workspace can share the interface of the parent window. For example, if the workspace includes a menu bar, the windows of any objects contained within the workspace share the menu bar. If the workspace does not have a menu bar, or if you provide an option for the user to hide the menu bar, the menu bar should appear within the document's child window. The parent window can also provide a framework for sharing toolbars and status bars.

## Window Management in a Workspace

A workspace manages windows using the same conventions as MDI. When a workspace closes, all the windows within it close. You should retain the state of these windows, for example, their size and position within the workspace, so you can restore them when the user reopens the workspace.

Like most primary windows, when the user minimizes the workspace window, the window disappears from the screen but its entry remains on the taskbar. Minimized windows of icons opened within the workspace have the same behavior and appearance as minimized MDI child windows. Similarly, maximizing a window within a workspace can follow the MDI technique: if the window's maximized size exceeds the size of the workspace window, the child window merges with the workspace window and its title bar icon and window buttons appear in the menu bar of the workspace window.

A workspace should provide a means of navigating between the child windows within a workspace, such as listing the open child windows on a Window drop-down menu and on the pop-up menu for the parent window, in addition to direct window activation.

## Workbooks

A *workbook* is another alternative for managing a set of views — one which uses the metaphor of a book or notebook instead of a work area. Within the workbook, you present views of objects as sections within the workbook's primary window rather than in individual child windows. Figure 9.5 illustrates one possible way of presenting a workbook.

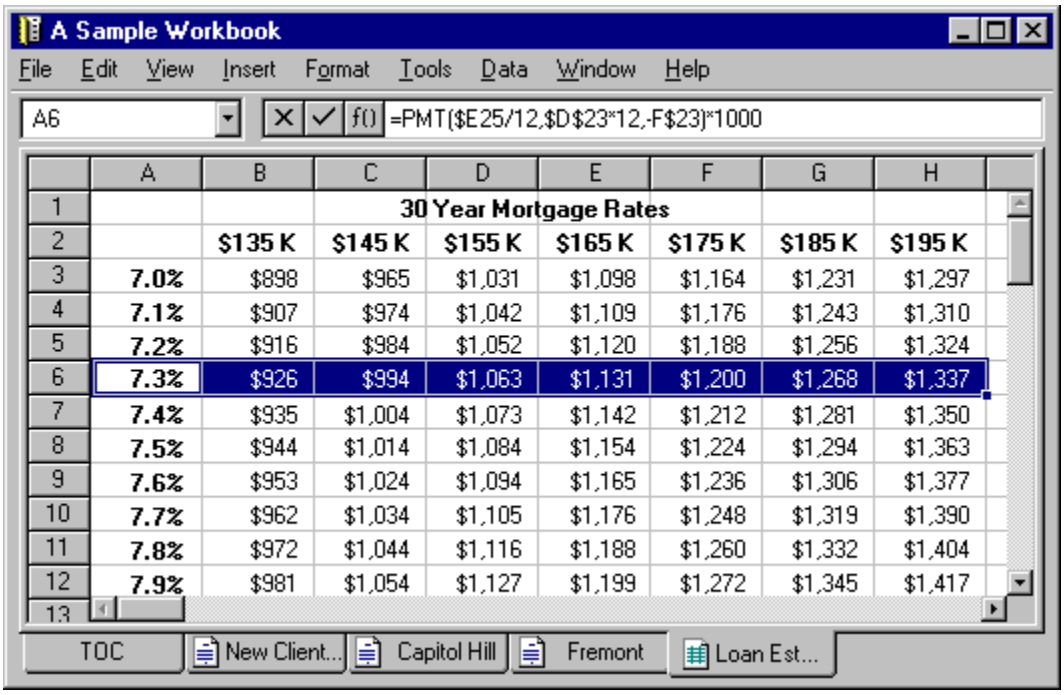



**Figure 9.5 Example of a workbook design**

For a workbook, you can use tabs to serve as a navigational interface to move between different sections. Locate the tabs as best fits the content and organization of the information you present. Each section represents a view of data, which could be an individual document. Unlike a folder or workspace, a workbook may be better suited for ordered content; that is, where the order of the sections has significance. In addition, you can optionally include a special section listing the content of the workbook, like a table of contents. This view can also be included as part of the navigational interface for the workbook.

A workbook shares an interface similar to an MDI parent window with all of its child windows maximized. The sections can share the parent window's interface elements, such as the menu bar and status bar. When the user switches sections within the workbook, you can change the menu bar so that it applies to the current object. When the user closes a workbook, follow the common conventions for handling unsaved edits or unapplied transactions when any primary window closes.

Consider supporting OLE to support transfer operations so the user can move, copy, and link objects into the workbook. You may also want to provide an Insert command that allows the user to create new objects, including a new tabbed section in the workbook. You can also include a Save All command, which saves any uncommitted changes or prompts the user to save or discard those changes.

## Projects

A *project* is another window management technique that provides for association of a set of objects and their windows, but without visually containing the windows. A project is similar to a folder in that the icons contained within it can be opened into windows that are peers with the parent window. As a result, each child window can also have its own entry on the taskbar. Unlike a folder, a project provides window management for the windows of its content. For example, when the user opens a document in a folder and then closes the folder, it has no effect on the window of the opened document. However, when the user closes a project window, all the child windows of objects contained in the project also close. In addition, when the user opens a project window, this action should restore the windows of objects contained within it to their previous state.

Similarly, to facilitate window management, when the user minimizes a project window, you may want to minimize any windows of the objects the project contains. Taskbar entries for these windows remain. Allow the user to restore a specific child window without restoring the project window or other windows within the project. In addition, support the user independently minimizing any child window without affecting the project window. Figure 9.6 shows an example of a project.

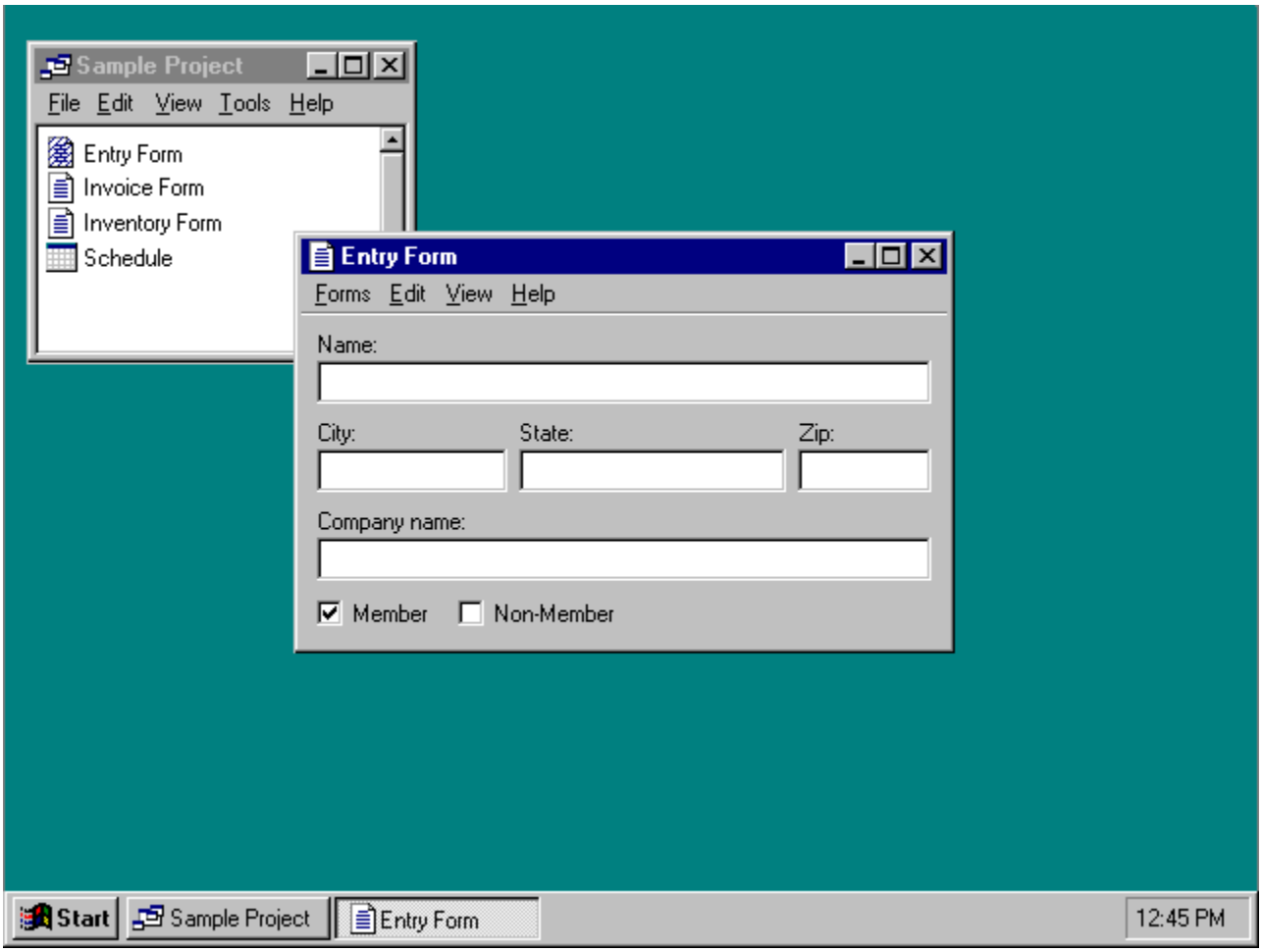



**Figure 9.6 Example of a project design**

The windows of objects stored in the project do not share the menu bar or other areas within the project window. Instead, include the interface elements for each object in its own window. However, you can provide toolbar palette windows that can be shared among the windows of the objects in the project.

Just as in workspaces and workbooks, a project should include commands for creating new objects within the project, for transferring objects in and out of the project, and for saving any changes for the objects stored in the project. In addition, a project should include commands and properties for the project object itself.

# Selecting a Window Model

Deciding how to present your application’s collection of related tasks or processes requires considering a number of design factors: your intended audience and their skill level, the presentation of object or task, effective use of the space on the display, and evolution towards data-centered design.

## Presentation of Object or Task

What an object represents and how it is used and relates to other objects influences how you present its view. Simple objects that are self-contained may not require a primary window, or only require a set of menu commands and a property sheet to edit their properties.

An object with user-accessible content in addition to properties, such as a document, only requires a primary window. The single document window interface can be sufficient when the object's primary presentation or use is as a single unit, even when containing different types. Alternative views can easily be supported with controls that allow the user to change the view. Simple, simultaneous views of the same data can even be supported by splitting the window into panes. The system uses the single document window style of interface for most of the components it includes, such as folders.

MDI, workspaces, workbooks, and projects are more effective when the composition of an object requires multiple views or the nature of the user's tasks requires views of multiple objects. These constructs provide a grouping and focus for a set of specific user activities, within the larger environment of the desktop.

MDI is best suited for viewing homogeneous types. The user cannot mix different objects within the same MDI parent windows unless you supply them as part of the application. On the other hand, you can use MDI to support simultaneous views of different objects.

Use a workbook when you want to optimize quick user navigation of multiple views. A workbook simplifies the task by eliminating the management of child windows, but in doing so, it limits the user's ability to see simultaneous views.

Workspaces and projects provide flexibility for viewing and mixing of objects and their windows. Use a workspace as you would MDI, when you want to clearly segregate the icons and their windows used in a task. Use a project when you do not want to constrain any child windows.

A project provides the greatest flexibility for user placement and arrangement of its windows. It does so, however, at the expense of an increase in complexity because it may be more difficult for a user to differentiate the child window of a project from windows of other applications.

## Display Layout

Consider the requirements for layout of information. For very high resolution displays, the use of menu bars, toolbars, and status bars poses little problem for providing adequate display of the information being viewed in a window. Similarly, the appearance of these common interface elements in each window has little impact on the overall presentation. At VGA resolution, however, this can be an issue. The interface components for a set of windows should not so dominate the user's work area that the user cannot easily view or manipulate their data.

MDI, workspaces, workbooks, and projects all allow some interface components to be shared among multiple views. Within shared elements, it must be clear when a particular interface component applies. Although you can automatically switch the content of those components, consider what functions are common across views or child windows and present them in a consistent way to provide for stability in the interface. For example, if multiple views share a Print toolbar button, present that button in a consistent location. If the button's placement constantly shifts when the user switches the view, the user's efficiency in performing the task may decrease. Note that shared interfaces may make user customization of interface components more complex because you need to indicate whether the customization applies to the current context or across all views.

Regardless of the window model you chose, always consider allowing users to determine which interface components they wish to have displayed. Doing so means that you also need to consider how to make basic functionality available if the user hides a particular component. For example, pop-up menus can often supplement the interface when the user hides the menu bar.

## Data-Centered Design

A single document window interface provides the best support for a simple, data-centered design and may be the easiest for users to learn; MDI supports a more conventional application-centered design. It is best suited to multiple views of the same data or contexts where the application does not represent views of user data. You can use workspaces, workbooks, and projects to provide single document window interfaces while preserving some of the management techniques provided by MDI.

## Combination of Alternatives

Single document window interfaces, MDIs, workspaces, workbooks, and projects are not exclusive design techniques. It may be advantageous to combine these techniques. For example, documents can be presented within a workspace. You can also design workbooks and projects as objects within a workspace. In similar fashion, a project might contain a workbook as one of its objects.

# Introduction

Users appreciate seamless integration between the system and their applications. This topic covers information about integrating your software with the system and how to extend its features, including using the registry to store information about your application, installing your application, using appropriate naming conventions, and supporting shell features, such as the taskbar, Control Panel, and Recycle Bin.

This topic is only intended to provide an overview. Details required for some conventions go beyond the scope of this guide. For information about these conventions, see the documentation included in the Microsoft Win32 Software Development Kit (SDK). In addition, some of these conventions and features may not be supported in all releases. For more information about specific releases, see Supporting Specific Versions of Windows.

# The Registry

Windows provides a special repository called the *registry* that serves as a central configuration database for user-, application-, and computer-specific information. Although the registry is not intended for direct user access, the information placed in it affects your appli- cation's user interface. Registered information determines the icons, commands, and other features displayed for files. The registry also makes it easier to manage and support configuration information used by your application and eliminates redundant information stored in different locations.

The registry is a hierarchical structure. Each node in the tree is called a key. Each key can contain subkeys and data entries called values. Key names cannot include a space, backslash (\), or wildcard character (* or ?). In the **HKEY_CLASSES_ROOT** key, names beginning with a period (**.**) are reserved for special syntax (filename extensions), but you can include a period within a key name. The name of a subkey must be unique with respect to its parent key. Key names are not localized into other languages, although their values may be.

The example registry entries in this topic represent only the hierarchical relationship of the keys. For more information about the registry and registry file formats, see the documentation included in the Win32 SDK.

A key can have any number of values. A value entry has three parts: the name of the value, its data type, and the value itself. Value entries larger than 2048 bytes should be stored as files with their filenames stored in the registry.

When the user installs your application, register keys for where application data is stored, for filename extensions, icons, shell commands, OLE registration data, and for any special extensions. To register your application's information, you can create a registration file and use the Registry Editor to merge this file into the system registry. You can also use other utilities that support this function, or use the system-supplied registry functions to access or manipulate registry data.

To use memory most efficiently, the system stores only the registry entries that have been installed and that are required for operation. Applications should never fail to write a registry entry because it is not already installed. To ensure this happens, use registry creation functions when adding an entry.

## Registering Application State Information

Use the registry to store state information for your application. Typically, the data you store here will be information you may have stored in initialization (.INI) files in previous releases of Windows. Create subkeys under the **Software** subkey in the **HKEY_LOCAL_MACHINE** and **HKEY_CURRENT_USER** keys that include information about your application.

**HKEY_LOCAL_MACHINE**
    **Software**
        ***CompanyName***
            ***ProductName***
                ***Version***
**...**
**HKEY_CURRENT_USER**
    **Software**
        ***CompanyName***
            ***ProductName***
                ***Version***

Use your application's **HKEY_LOCAL_MACHINE** entry as the location to store computer-specific data and the **HKEY_CURRENT _USER** entry to store user-specific data. The latter key allows you to store settings to tailor your application for individual users working with the same computer. Under your application's subkey, you can define your own structure for the information. Although the system still supports initialization files for backward compatibility, use the registry wherever possible to store your application's state information instead.

Use these keys to save your application's state whenever appropriate, such as when the user closes its primary window. In most cases, it is best to restore a window to its previous state when the user reopens it.

When the user shuts down the system with your application's window open, you may optionally store information in the registry so that the application's state is restored when the user starts up Windows. (The system does this for folders.) To have your application's state restored, store your window and application state information under its registry entries when the system notifies your application that it is shutting down. Store the state information in your application's entries under **HKEY_CURRENT_USER** and add a value name–value pair to the **RunOnce** subkey that corresponds to your application. When the user restarts the system, it runs the command line you supply. Once your application runs, you can use the data you stored to restore its state.

**HKEY_CURRENT_USER**
    **Software**
        **Microsoft**
            **Windows**
                **CurrentVersion**
                    **RunOnce** *application identifier = command line*

If you have multiple instances open, you can include value name entries for each or consolidate them as a single entry and use command-line switches that are most appropriate for your application. For example, you can include entries like the following.

WordPad Document 1 = C:\Program Files\Wordpad.exe Letter to Bill /restore
WordPad Document 2 = C:\Program Files\Wordpad.exe Letter to Paul /restore
Paint = C:\Program Files\Paint.exe Abstract.bmp Cubist.bmp

As long as you provide a valid command-line string that your application can process, you can format the entry in a way that best fits your application.

You can also include a **RunOnce** entry under the **HKEY_LOCAL_ MACHINE** key. When using this entry, however, the system runs the application before starting up. You can use this entry for applications that may need to query the user for information that affects how Windows starts. Just remember that any entry here will affect all users of the computer.

**RunOnce** entries are automatically removed from the registry once the system starts up. Therefore,

you need not remove or update the entries, but your application must always save its state when the user shuts down the system. The system also supports a **Run** subkey in both the **HKEY_CURRENT_USER** and **HKEY_LOCAL_ MACHINE** keys. The system runs any value name entries under this subkey after the system starts up, but does not remove those entries from the registry. For example, a virus check program can be installed to run automatically after the system starts up. You can also support this functionality by placing a file or shortcut to a file in the Startup folder. The registry stores the location of the Startup folder, as a value in **HKEY_CURRENT_USER \Software\Microsoft\ Windows\CurrentVersion\Explorer\Shell Folders**.

The system's ability to restore an application's state depends on the availability of the application and its data files. If they have been deleted or the user has logged in over the network where the same files are not available, the system may not be able to restore the state.

## Registering Application Path Information

The system supports “per application” paths. If you register a path, Windows sets the PATH environment variable to be the registered path when it starts your application. You set your application’s path in the **App Paths** subkey under the **HKEY_LOCAL_MACHINE** key. Create a new key using your application’s executable filename as its name. Set this key’s Default value to the path of your executable file. The system uses this entry to locate your application if it fails to find it in the current path; for example, if the user chooses the Run command on the Start menu and only includes the filename of the application, or if a shortcut icon doesn’t include a path setting. To identify the location of dynamic-link libraries placed in a separate directory, you can also include another value entry called Path and set its value to the path of your dynamic-link libraries.

**HKEY_LOCAL_MACHINE**
**Software**
**Microsoft**
**Windows**
**CurrentVersion**
**App Paths**
***Application Executable Filename*** = *path*
Path = *path*

The system will automatically update the path and default entries if the user moves or renames the application’s executable file using the system shell user interface.

Register any system-wide shared dynamic-link libraries in a subkey under a **SharedDLLs** subkey of **HKEY_LOCAL_MACHINE** key. If the file already exists, increment the entry’s usage count index. For more information about the usage count index, see Installation.

**HKEY_LOCAL_MACHINE**
**Software**
**Microsoft**
**Windows**
**CurrentVersion**
**SharedDLLs** *filename* [= *usage count index*]

## Registering File Extensions

If your application creates and maintains files, register entries for the file types that you expose directly to users and that you want users to be able to easily differentiate. For every file type you register, include at least two entries: a filename-extension key entry and an application (class) identification key entry.

If you do not register an extension for a file type, it will be displayed with the system's generic file object icon, as shown in Figure 10.1, and its extension will always be displayed. In addition, the user will not be able to double-click the file to open it. (Open With will be the icon's default command.)

**Figure 10.1 System-generated icons for unregistered types**

## The Filename Extension Key

The filename extension entry maps a filename extension to an application identifier. To register an extension, create a subkey in the **HKEY_CLASSES_ROOT** key using the three-letter extension (including a period) and set its value to an application identifier.

**HKEY_CLASSES_ROOT**
***.ext*** = *ApplicationIdentifier*

For the value of the application identifier (also known as programmatic identifier or Prog ID), use a string that uniquely identifies a given class. This string is used internally by the system and is not exposed directly to users (unless explicitly exported with a special registry utility); therefore, you need not localize this entry.

Avoid assigning multiple extensions to the same application identifier. To ensure that each file type can be distinguished by the user, define each extension such that each has a unique application identifier. If you have utility files that the user does not interact with directly, you should still register an extension (and icon) for them, preferably the same extension so that they can be identified. In addition, mark them with the hidden file attribute.

The system provides no arbitration for applications that use the same extensions. So define unique identifiers and check the registry to avoid writing over and replacing existing extension entries, a practice which may seriously affect the user's existing files. More specifically, avoid registering an extension that conflicts or redefines the common filename extensions used by the system. Examples of these extensions are shown in Table 10.1.

**Table 10.1 Common Filename Extensions Supported by Windows**

| Extension | Type description |
|---|---|
| 386 | Windows virtual device driver |
| 3GR | Screen grabber for MS-DOS–based applications |
| ACM | Audio compression manager driver |
| ADF | Administration configuration files |
| ANI | Animated pointer |
| AVI | Video clip |
| AWD | FAX viewer document |
| AWP | FAX key viewer |
| AWS | FAX signature viewer |
| BAK | Backed-up file |
| BAT | MS-DOS batch file |
| BFC | Briefcase |
| BIN | Binary data file |
| BMP | Picture (Windows bitmap) |
| CAB | Windows Setup file |
| CAL | Windows Calendar file |
| CDA | CD audio track |
| CFG | Configuration file |
| CNT | Help contents |
| COM | MS-DOS-based application |
| CPD | FAX cover page |
| CPE | FAX cover page |
| CPI | International code page |

| | |
|---|---|
| CPL | Control Panel extension |
| CRD | Windows Cardfile document |
| CSV | Command-separated data file |
| CUR | Cursor (pointer) |
| DAT | System data file |
| DCX | FAX viewer document |
| DLL | Application extension (dynamic-link library) |
| DOC | WordPad document |
| DOS | MS-DOS file (also extension for NDIS2 net card and protocol drivers) |
| DRV | Device driver |
| EXE | Application |
| FND | Saved search |
| FON | Font file |
| FOT | Shortcut to font |
| GR3 | Windows 3.0 screen grabber |
| GRP | Program group file |
| HLP | Help file |
| HT | HyperTerminal™ file |
| ICM | ICM profile |
| ICO | Icon |
| IDF | MIDI instrument definition |
| INF | Setup information |
| INI | Initialization file (configuration settings) |
| KBD | Keyboard layout |
| LGO | Windows logo driver |
| LIB | Static-link library |
| LNK | Shortcut |
| LOG | Log file |
| MCI | MCI command set |
| MDB | File viewer extension |
| MID | MIDI sequence |
| MIF | MIDI instrument file |
| MMF | Microsoft Mail message file |
| MMM | Animation |
| MPD | Mini-port driver |
| MSG | Microsoft® Exchange mail document |
| MSN | Microsoft Network home base |
| NLS | Natural language services driver |
| PAB | Microsoft Exchange personal address book |
| PCX | Bitmap picture (PCX format) |
| PDR | Port driver |
| PF | ICM profile |
| PIF | Shortcut to MS-DOS–based application |
| PPD | PostScript® printer description file |

| | |
|---|---|
| PRT | Printer formatted file (result of Print to File option) |
| PST | Microsoft Exchange personal information store |
| PWL | Password list |
| QIC | Backup set for Microsoft Backup |
| REC | Windows Recorder file |
| REG | Application registration file |
| RLE | Picture (RLE format) |
| RMI | MIDI sequence |
| RTF | Document (rich-text format) |
| SCR | Screen saver |
| SET | File set for Microsoft Backup |
| SHB | Shortcut into a document |
| SHS | Scrap |
| SPD | PostScript printer description file |
| SWP | Virtual memory storage |
| SYS | System file |
| TIF | Picture (TIFF® format) |
| TMP | Temporary file |
| TRN | Translation file |
| TSP | Windows telephony service provider |
| TTF | TrueType® font |
| TXT | Text document |
| VBX | Microsoft Visual Basic® control file |
| VER | Version description file |
| VXD | Virtual device driver |
| WAV | Sound wave |
| WPC | WordPad file converter |
| WRI | Windows Write document |

It is a good idea to investigate extensions commonly used by popular applications so you can avoid creating a new extension that might conflict with them, unless you intend to replace or superset the functionality of those applications.

## The Application Identifier Key

The second registry entry you create for a file type is its class-definition (Prog ID) key. Using the same string as the application identifier you used for the extension's value, create a key, and assign a type name as the value of the key.

**HKEY_CLASSES_ROOT**
***.ext*** = *ApplicationIdentifier*
***ApplicationIdentifier*** = *Type Name*

Under this key, you specify shell and OLE properties of the class. Provide this entry even if you do not have any extra information to place under this key; doing so provides a label for users to identify the file type. In addition, you use this entry to register the icon for the file type.

Define the type name (also known as the MainUserTypeName) as the human-readable form of its application identifier or class name. It should convey to the user the object's name, behavior, or capability. A type name can include all of the following elements:

1. *Company Name*
   Communicates product identity.
2. *Application Name*
   Indicates which application is responsible for activating a data object.
3. *Data Type*
   Indicates the basic category of the object (for example, drawing, spreadsheet, or sound). Limit the number of characters to a maximum of 15.
4. *Version*
   When there are multiple versions of the same basic type, for upgrading purposes, you may want to include a version number to distinguish types.

When defining your type name, use title capitalization. The name can include up to a maximum of 40 characters. Use one of the following three recommended forms:

1. *Company Name Application Name* [*Version*] *Data Type*
   For example, Microsoft Excel Worksheet.
2. *Company Name-Application Name* [*Version*] *Data Type*
   For cases when the company name and application are the same — for example, ExampleWare 2.0 Document.
3. *Company Name Application Name* [*Version*]
   When the application sufficiently describes the data type — for example, Microsoft Graph.

These type names provide the user with a precise language for referring to objects. Because object type names appear throughout the interface, the user becomes conscious of an object's type and its associated behavior. However, because of their length, you may also want to include a short type name. A *short type name* is the data type portion of the full type name. Applications that support OLE always include a short type name entry in the registry. Use the short type name in drop-down and pop-up menus. For example, a Microsoft® Excel Worksheet is simply referred to as a "Worksheet" in menus.

To provide a short type name, add an **AuxUserType** subkey under the application's registered **CLSID** subkey (which is under the **CLSID** key).

For more information about registering type names and other information you should include under the **CLSID** key, see the OLE documentation included in the Win32 SDK.

**HKEY_CLASSES_ROOT**
***.ext*** = *ApplicationIdentifier*
...
***ApplicationIdentifier*** = *Type Name*
**CLSID** = **{*CLSID identifier*}**
...

**CLSID**
**{*CLSID identifier*}**
**AuxUserType**
**2** = *Short Type Name*

If a short type name is not available for an object because the string was not registered, use the full type name instead. All controls that display the full type name must allocate enough space for 40 characters in width. By comparison, controls need only accommodate 15 characters when using the short type name.

## Supporting Creation

The system supports the creation of new objects in system containers, such as folders and the desktop. Register information for each file type that you want the system to include. The registered type will appear on the New command that the system includes on menus for the desktop, folders, and the Open and Save As common dialog boxes. This provides a more data-centered design because the user can create a new object without having to locate and run the associated application.

To register a file type for inclusion, create a subkey using the Application Identifier under the extension's subkey in **HKEY_ CLASSES_ROOT**. Under it, also create the **ShellNew** subkey.

**HKEY_CLASSES_ROOT**
    ***.ext*** = *ApplicationIdentifier*
        ***ApplicationIdentifier***
            **ShellNew** *Value Name* = *Value*

Assign a value entry to the **ShellNew** subkey with one of the four methods for creating a file with this extension.

| Value name | Value | Result |
|---|---|---|
| NullFile | " " | Creates a new file of this type as a null (empty) file. |
| Data | *binary data* | Creates a new file containing the binary data. |
| FileName | *path* | Creates a new file by copying the specified file. |
| Command | *filename* | Carries out the command. Use this to run your own application code to create a new file (for example, run a wizard). |

The system also will automatically provide a unique filename for the new file using the type name you register.

When using a Command value, place your application file (that creates the new file) in the directory that the system uses to store these files. To determine the path for that directory, check the setting for the Templates value in the **Shell Folders** subkey found in **HKEY_CURRENT_USER\Software\Microsoft\Windows\ CurrentVersion\Explorer**. Then you need only register the filename for the command.

## Registering Icons

The system uses the registry to determine which icon to display for a specific file. You register an icon for every data file type that your application supports and that you want the user to be able to distinguish easily. Create a **DefaultIcon** subkey entry under the application identifier subkey you created and define its value as the filename containing the icon. Typically, you use the application's executable filename and the index of the icon within the file. The index value corresponds to the icon resource within the file. A positive number represents the icon's position in the file. A negative number corresponds to the inverse of the resource ID number of the icon. The icon for your application should always be the first icon resource in your executable file. The system always uses the first icon resource to represent executable files. This means the index value for your data files will be a number greater than 0.

**HKEY_CLASSES_ROOT**
***ApplicationIdentifier*** = *Type Name*
**DefaultIcon** = *path* [,*index*]

Instead of registering the application's executable file, you can register the name of a dynamic link library file (.DLL), an icon file (.ICO), or bitmap file (.BMP) to supply your data file icons. If an icon does not exist or is not registered, the system supplies an icon derived from the icon of the file type's registered application. If no icon is available for the application, the system supplies a generic icon. These icons do not make your files uniquely identifiable, so design and register icons for both your application and its data file types. Include the following sizes: 16 x 16 pixel (16 color), 32 x 32 pixel (16 color), and 48 x 48 pixel (256 color).

For more information about designing icons, see Visual Design.

## Registering Commands

Many of the commands found on icons, including Send To, Cut, Copy, Paste, Create Shortcut, Delete, Rename, and Properties, are provided by their container — that is, their containing folder or the desktop. But you must provide support for the icon's primary commands, also referred to as verbs, such as Open, Edit, Play, and Print. You can also register additional commands that apply to your file types, such as a What's This? command and even commands for other file types.

To add these commands, in the **HKEY_CLASSES_ROOT** key, you register a **shell** subkey and a subkey for each verb, and a **command** subkey for each menu command name.

**HKEY_CLASSES_ROOT**
    ***ApplicationIdentifier*** = *Type Name*
        **shell** [ = *default verb* [,*verb2* [,..]]
            ***verb*** [ = *Menu Command Name*]
                **command** = *pathname* [*parameters*]

You can also register a DDE command string for a DDE command.

**HKEY_CLASSES_ROOT**
    ***ApplicationIdentifier*** = *Type Name*
        **shell** [ = *default verb* [,*verb2* [,..]]
            ***verb*** [ = *Menu Command Name*]
                **ddeexec** = *DDE command string*
                    **Application** = *DDE Application Name*
                    **Topic** = *DDE topic name*

A verb is a language-independent name of the command. Applications may use it to invoke a specific command programmatically. The system defines Open, Print, Find, and Explore as standard verbs and automatically provides menu command names and appropriate access key assignments, localized in each international version of Windows. When you supply verbs other than these, provide menu command names localized for the specific version of Windows on which the application is installed. To assign a menu command name for a verb, make it the default value of the verb subkey.

The menu command names corresponding to the verbs for a file type are displayed to the user, either on a folder's File drop-down menu or pop-up menu for a file's icon. These appear at the top of the menu. You define the order of the menu commands by ordering the verbs in the value of the **shell** key. The first verb becomes the default command in the menu.

By default, capitalization follows how you enter format the menu command name value of the verb subkey. Although the system automatically capitalizes the standard commands (Open, Print, Explore, and Find), you can use the value of the menu command name to format the capitalization differently. Similarly, you use the menu command name value to set the access key for the menu command following normal menu conventions, prefixing the character in the name with an ampersand (&). Otherwise, the system sets the first letter of the command as the access key for that command.

To support user execution of a verb, provide the path for the application or a DDE command string. You can include command-line switches. For paths, include a %1 parameter. This parameter is an operational placeholder for whatever file the user selects.

For example, to register an Analyze command for an application that manages stock market information, the registry entries might look like the following.

**HKEY_CLASSES_ROOT**
    **stockfile** = Stock Data
        **shell** = analyze
            **analyze** = &Analyze
                **command** = C:\Program Files\Stock Analysis\Stock.exe /A

You may have different values for each command. You may assign one application to carry out the Open command and another to carry out the Print command, or use the same application for all commands.

## Enabling Printing

If your file types are printable, include a Print verb entry in the **shell** subkey under **HKEY_CLASSES_ROOT**, following the conventions described in the previous section. This will display the Print command on the pop-up menu for the icon and on the File menu of the folder in which the icon resides when the user selects the icon. When the user chooses the Print command, the system uses the registry entry to determine what application to run to print the file.

Also register a Print To registry entry for the file types your application supports. This entry enables dragging and dropping of a file onto a printer icon. Although a Print To command is not displayed on any menu, the printer includes Print Here as the default command on the pop-up menu displayed when the user drag and drops a file on the printer using button 2.

In both cases, print the file, preferably, without opening the application's primary window. One way to do this is to provide a command-line switch that runs the application for handling the printing operation only (for example, WordPad.exe /p). In addition, display some form of user feedback that indicates whether a printing process has been initiated and, if so, its progress. For example, this feedback could be a modeless message box that displays, "Printing page *m* of *n* on *printer name*" and a Cancel button. You may also include a progress indicator control.

## Registering OLE

Applications that support OLE use the registry as the primary means of defining class types, operations, and properties for data types supported by those applications. You store OLE registration information in the **HKEY_CLASSES_ROOT** key in subkeys under the **CLSID** subkey and in the class description's (Prog ID) subkey.

For more information about the specific registration entries for OLE, see the OLE documentation included in the Win32 SDK.

## Registering Shell Extensions

Your application can extend the functionality of the operational environment provided by the system, also known as the shell, in a number of ways. A shell extension enhances the system by providing additional ways to manipulate file objects, by simplifying the task of browsing through the file system, or by giving the user easier access to tools that manipulate objects in the file system.

Support for shell extensions may depend on the version of Windows installed. For more information about specific releases, see Supporting Specific Versions of Windows.

Every shell extension requires a *handler*, special application code (32-bit OLE InProc server implemented as a dynamic-link library) that implements subordinate functions. The types of handlers you can provide include:

- Pop-up (context) menu handlers: these add menu items to the pop-up menu for a particular file type.
- Drag handlers: these allow you to support the OLE data transfer conventions for drag and drop operations of a specific file type.
- Drop handlers: these allow you to carry out some action when the user drops objects on a specific type of file.
- Nondefault drag and drop handlers: these are pop-up menu handlers that the system calls when the user drags and drops an object by using mouse button 2.
- Icon handlers: these can be used to add per-instance icons for file objects or to supply icons for all files of a specific type.
- Property sheet handlers: these add pages to a property sheet that the shell displays for a file object. The pages can be specific to a class of files or to a particular file object.
- Copy-hook handlers: these are called when a folder or printer object is about to be moved, copied, deleted, or renamed by the user. The handler can be used to allow or prevent the operation.

You register the handler for a shell extension in the **HKEY_ CLASSES_ROOT** key. The **CLSID** subkey contains a list of class identifier key values such as {00030000-0000-0000- C000-000000000046}. Each class identifier must also be a globally unique identifier.

For more information about creating handlers and class identifiers, see the OLE documentation included in the Win32 SDK.

You must also create a **shellex** subkey under the application's class identification entry in the **HKEY_CLASSES_ROOT** key.

**HKEY_CLASSES_ROOT**
    ***ApplicationIdentifier*** = *Type Name*
        **Shell** [ = *default verb* [,*verb2* [,..]]
            ...
            **shellex**
                ***HandlerType***
                    **{*CLSID identifier*}** = *Handler Name*
                ...
                ***HandlerType*** = {*CLSID identifier*}

The shell also uses several other special keys, such as *****, **Folder**, **Drives**, and **Printers**, under **HKEY_CLASSES_ROOT**. You can use these keys to register extensions for system-supplied objects. For example, you may use the ***** key to register handlers that the shell calls whenever it creates a pop-up menu or property sheet for a file object, as in the following example.

**HKEY_CLASSES_ROOT**
    ***** = *
        **shellex**
            **ContextMenuHandlers**

**{00000000-1111-2222-3333-0000000001}**
**PropertySheetHandlers** = SummaryInfo
**{00000000-1111-2222-3333-0000000002}**
**IconHandler =** {00000000-1111-2222-3333-000000003}

The shell would use these handlers to add to the pop-up menus and property sheets of every file object. (The entries are intended only as examples, not literal entries.)

A pop-up menu handler may add commands to the pop-up menu of a file type, but it may not delete or modify existing menu commands. You can register multiple pop-up menu handlers for a file type. The order of the subkey entries determines the order of the items in the context menu. Handler-supplied menu items always follow registered command names.

Keep in mind that if you want to include a command on the pop-up menu of every file of a particular type, you do not need to create and register a pop-up menu handler. You can just use the normal means of registering commands for that type. Create a pop-up menu handler only when you want to provide a command only under specific conditions, such as the length of the file or its timestamp.

When registering an icon handler for providing per-instance icons for a file type, set the value for the **DefaultIcon** key to %1. This denotes that each file instance of this type can have a different icon.

## Supporting the Quick View Command

The system includes support for fast, read-only views of many file types when the user chooses the Quick View command from the file object's menu. This allows the user to view files without opening the application.

For more information about supporting the Quick View command and creating file viewers, see the documentation included in the Win32 SDK.

If your file type is not supported, you can install a file parser that translates your file type into a format the system file viewer can read. Although this approach allows you to easily support viewers for your data file types, it limits the interaction options for your file types to those provided by the system. Alternatively, you can create your own file viewer, using the system-supplied interfaces. You can also register a file viewer for a file type already registered.

You can also support the Quick View command for data objects stored within your application's interface, either by supplying a specific viewer for your data types or by writing the data to a temporary file and then executing a file viewer and passing the temporary file as a parameter.

## Registering Sound Events

Your application can register specific events to which the user can assign sound files so that when those events are triggered, the assigned sound file is played. To register a sound event, create a key under the **HKEY_CURRENT_USER** key.

**HKEY_CURRENT_USER**
**AppEvents**
**Event Labels**
***EventName*** = *Event Name*

Set the value for EventName to a human-readable name.

Registering a sound event only makes it available in Control Panel so the user can assign a sound file. Your application must provide the code to process that event.

## Installation

The following sections provide guidelines for installing your application's files. Applying these guidelines will help you reduce the clutter of irrelevant files when the user browses for a file. In addition, you'll reduce the redundancy of common files and make it easier for the user to update applications or the system software.

## Copying Files

When the user installs your software, avoid copying files into the Windows directory (folder) or its System subdirectory. Doing so clutters the directory and may degrade system performance. Instead, create a single directory, preferably using the application's name, in the Program Files directory (or the location that the user chooses). In this directory, place the executable file. For example, if a program is named My Application, create a My Application subdirectory and place My Application.exe in that directory.

To locate the Program Files directory, check the ProgramFilesDir value in the **CurrentVersion** subkey under **HKEY_LOCAL_ MACHINE\Software\Microsoft\Windows**. The actual directory may not literally be named Program Files. For example, in international versions of Microsoft Windows, the directory name is appropriately localized. For networks that do not support the Windows long filename conventions, MS-DOS names may be used instead.

In your application's directory, create a subdirectory named System and place all support files that the user does not directly access in it, such as dynamic-link libraries and Help files. For example, place a support file called My Application.dll in the subdirectory Program Files\My Application\System. Hide the support files and your application's System directory and register its location using a Path value in the **App Paths** subkey under **HKEY_LOCAL_MACHINE \Software\Microsoft\Windows\CurrentVersion**. Although you may place support files in the same directory as your application, placing them in a subdirectory helps avoid confusing the user and makes files easier to manage.

Applications can share common support files to reduce the amount of disk space consumed by duplication. If some non-user-accessed files of your application are shared as systemwide components (such as Visual Basic's Vbrun300.dll), place them in the System subdirectory of the directory where the user installs Windows. The process for installing shared files includes these logical steps:

The system provides support services in Ver.dll for assisting you to do version verification. For more information about this utility, see the documentation included in the Win32 SDK.

1. Before copying the file, verify that it is not already present.
2. If the file is already present, compare its date and size to determine whether it is the same version as the one you are installing. If it is, increment the usage count in its corresponding registry entry.
3. If the file you are installing is not more recent, do not overwrite the existing version.
4. If the file is not present, copy it to the directory.

If you store a new file in the System directory installed by Windows, register a corresponding entry in the **SharedDLL** subkey under the **HKEY_LOCAL_MACHINE** key.

If a file is shared, but only among your applications, create a subdirectory using your application's name in the Common Files subdirectory of the Program Files subdirectory and place the file there. To locate the Common Files directory, check the Common-FilesDir value in the **CurrentVersion** subkey of **HKEY_LOCAL_ MACHINE\Software\Microsoft\Windows**. Alternatively, for "suite" style when multiple applications are bundled together, you can create a suite subdirectory in Program Files, where you place your executable files, and within that a System subdirectory with the support files shared only within the suite. In either case, register the path using the **Path** subkey under the **App Paths** subkey

When installing an updated version of the shared file, ensure that it is upwardly compatible before replacing the existing file. Alternatively, you can create a separate entry with a different filename (for example, Vbrun301.dll).

Name your executable file, dynamic-link libraries, and any other files that the user does not directly use, but that may be shared on a network, using conventional MS-DOS (8.3) names rather than long filenames. This will provide better support for users operating in environments where these files may need to be installed on network services that do not support the Windows long filename conventions.

Windows no longer requires Autoexec.bat and Config.sys files. Ensure that your application also does not require these files. Consider converting any MS-DOS device drivers to Windows virtual device drivers. The system supports dynamic loading of this type of device drivers, unlike MS-DOS device

drivers which need to be loaded through Config.sys when starting the system. Similarly, because the registry allows you to register your application paths, your application does not require path information in Autoexec.bat.

In addition, do not make entries in Win.ini. Storing information in this file can make it difficult for the user to update or move your application. Also, avoid maintaining your application's own initialization file. Instead, use the registry. The registry provides conventions for storing most application and user settings. The registry provides greater flexibility allowing you to store information on a per machine or per user basis. It also supports accessing this information across a network.

Make certain you register the types supported by your application and the icons for these types along with your application's icons. In addition, register other application information, such as information required to enable printing.

## Providing Access to Your Application

To provide easy user access to your application, place a shortcut icon to the application in the Programs folder. You can determine the path for this folder in **Shell Folders** subkey under **HKEY_CURRENT_ USER\Software\Microsoft\Windows\CurrentVersion\Explorer**. This adds the entry to the submenu of the Programs menu of the Start button. Avoid adding entries for every application you might include in your software; this quickly overloads the menu. Optionally, you can allow the user to choose which icons to place in the menu. Avoid using a folder as your entry in the Programs menu, because this creates a multilevel hierarchy. Including a single entry makes it easier and simpler for a user to access your application.

You can create a “program group” entry in the Programs folder using the Windows 3.1 dynamic date exchange application programming interface (API). However, it is not recommended for applications installed with Microsoft Windows 95 and later releases, configured with the new shell user interface.

Also consider the layout of files you provide with your application. Folders in Windows 95 and later releases provide much greater flexibility for file organization than did the Windows Program Manager. In addition to the recommended structure for your main executable file and its support files, you may want to create special folders for documents, templates, conversion tools, or other files that the user accesses directly.

## Designing Your Installation Program

Your installation program should offer the user different installation options such as:

- Typical Setup: installation that proceeds with the common defaults set, copying only the most common files. Make this the default setup option.
- Compact Setup: installation of the minimum files necessary to operate your application. This option is best for situations where disk space must be conserved — for example, on laptop computers. You can optionally add a Portable setup option for additional functionality designed especially for configurations on laptops, portables, and portables used with docking stations.
- Custom Setup: installation for the experienced user. This option allows the user to choose where to copy files and which options or features to include. This can include options or components not available for compact or typical setup.
- CD-ROM Setup: installation from a CD-ROM. This option allows users to select what files to install from the CD and allows them to run the remaining files directly from the CD.
- Silent Setup: installation using a command-line switch. This allows your setup program to run with a batch file

In addition to these setup options, your installation program should be a well-designed, Windows-based application and follow the conventions detailed in this guide and in the following guidelines:

- Supply a common response to every option so that the user can step through the installation process by confirming the default settings (that is, by pressing the ENTER key).
- Tell users how much disk space they will need before proceeding with installation. In the custom setup option, adjust the figure as the user chooses to include or exclude certain options. If there is not sufficient disk space, let the user know, but also give the user the option to override.
- Offer the user the option to quit the installation before it is finished. Keep a log of the files copied and the settings made so the canceled installation can be cleaned up easily.
- Ask the user to insert a disk only once during the installation. Lay out your files on disk so that the user does not have to reinsert the same disk multiple times.
- Provide a visual prompt and an audio cue when the user needs to insert the next disk.
- Support installation from any location. Do not assume that installation must be done from a logical MS-DOS drive (such as drive A). Design your installation program to support any valid universal naming convention (UNC) path.
- Provide a progress indicator message box to inform the user how far they are through the installation process.

If you are creating your own installation program, consider using the wizard control. Using this control and following the guidelines for wizards will result in a consistent interface for users.

For more information about designing wizards, see User Assistance.

Naming your installation program Setup.exe or Install.exe (or localized equivalent) will allow the system to recognize the file. Place the file in the root directory of the disk the user inserts. This allows the system to automatically run your installation program when the user chooses the Install button in the Add/Remove Programs utility in Control Panel.

## Installing Fonts

When installing fonts with your application on a local system, determine whether the font is already present. If it is, rename your font file — for example, by appending a number to the end of its filename. After copying a font file, register the font in the **Fonts** subkey.

## Installing Your Application on a Network

If you create a client-server application so that multiple users access it from a network server, create separate installation programs: an installation program that allows the network administrator to prepare the server component of the application, and a client installation program that installs the client component files and sets up the settings to connect to the server. Design your client software so that an administrator can deploy it over the network and have it automatically configure itself when the user starts it.

For more information about designing client-server applications, see Special Design Considerations. Additional information can also be found in the documentation included in Win32 SDK.

Because Windows may itself be configured to be shared on a server, do not assume that your installation program can store information in the main Windows directory on the server. In addition, shared application files should not be stored in the “home” directory provided for the user.

Design your installation program to support UNC paths. Also, use UNC paths for any shortcut icons you install in the Start Menu folder.

## Uninstalling Your Application

The user may need to remove your application to recover disk space or to move the application to another location. To facilitate this, provide an uninstall program with your application that removes its files and settings. Remember to remove registry entries and shortcuts your application may have placed in the Start menu hierarchy. However, be careful when removing your application's directory structure not to delete any user files (unless you confirm their removal with the user).

Your uninstall program should follow the conventions detailed in this guide and in the following guidelines:

- Display a window that provides the user with information about the progress of the uninstall process. You can also provide an option to allow the program to uninstall "silently" — that is, without displaying any information so that it can be used in batch files.
- Display clear and helpful messages for any errors your uninstall program encounters during the uninstall process.
- When uninstalling an application, decrement the usage count in the registry for any shared component — for example, a dynamic-link library. If the result is zero, give the user the option to delete the shared component with the warning that other applications may use this file and will not work if it is missing.

Registering your uninstall program will display your application in the list of the Uninstall page of the Add/Remove Program utility included with Windows. To register your uninstall program, add the entries for your application to the **Uninstall** subkey.

**HKEY_LOCAL_MACHINE**
**Software**
**Microsoft**
**Windows**
**CurrentVersion**
**Uninstall**
***ApplicationName*** DisplayName = *Application Name*
UninstallString = *path* [ *switches* ]

Both the **DisplayName** and **UninstallString** values must be supplied and be complete for your uninstall program to appear in the Add/Remove Program utility. The path you supply to **Uninstall-String** must be the complete command line used to carry out your uninstall program. The command line you supply should carry out the uninstall program directly rather than from a batch file or subprocess.

## Supporting AutoPlay

Windows supports the ability to automatically run a file when the user inserts removable media that support insertion notification, such as CD-ROM, PCMCIA hard disks, or flash ROM cards. To support this feature, include a file named Autorun.inf in the root directory of the removable media. In this file, include the filename of the file to run, using the following syntax.

[autorun]
open = *filename*

Unless you specify a path, the system looks for the file in the root of the inserted media. If you want to run a file located in a subdirectory, include a path relative to the root; include that path with the file as in the following example.

open = My Directory\My File.exe

Running the file from a subdirectory does not change the current directory setting. The command-line string you supply can also include parameters or switches.

Because the autoplay feature is intended to provide automatic operation, design the file you specify in the Autorun.inf file to provide visual feedback quickly to confirm the successful insertion of the media. Consider using a startup up window with a graphic or animated sequence. If the process you are automating requires a long load time or requires user input, offer the user the option to cancel the process.

Although you can use this feature to install an application, avoid writing files to the user's local disk without the user's confirmation. Even when you get the user's confirmation, minimize the file storage requirements, particularly for CD-ROM games or educational applications. Consuming a large amount of local file space defeats some of the benefits of the turnkey operation that the autoplay feature provides. Also, because a network administrator or the user can disable this feature, avoid depending on it for any required operations.

You can define the icon that the system displays for the media by including an entry in the Autorun.inf file that includes the filename (and optionally the path) including the icon using the following form.

icon = *filename*

The filename can specify an icon, a bitmap, an executable, or a dynamic-link library file. If the file contains more than one icon resource, specify the resource with a number after the filename — for example, My File.exe, 1. The numbering follows the same conventions as the registry. The default path for the file will be relative to the Autorun.inf file. If you want to specify an absolute path for an icon, use the following form.

defaulticon = *path*

The system automatically provides a pop-up menu for the icon and includes AutoPlay as the default command on that menu, so that double-clicking the icon will run the Open = line. You can include additional commands on the menu for the icon by adding entries for them in the Autorun.inf file, using the following form.

shell\verb\command = *filename*

shell\verb = *Menu Item Name*

To define an access key assignment for the command, precede the character with an ampersand (&). For example, to add the command Read Me First to the menu of the icon, include the following in the Autorun.inf file.

shell\readme\command = Notepad.exe My Directory\Readme.txt

shell\readme = Read &Me First

Although AutoPlay is typically the default menu item, you can define a different command to be the default by including the following line.

shell = verb

When the user double-clicks on the icon, the command associated with this entry will be carried out.

## System Naming Conventions

Windows provides support for filenames up to 255 characters long. Use the long filename when displaying the name of a file. Avoid displaying the filename extension unless the user chooses the option to display extensions or when the file type is not registered.

The system automatically formats a filename correctly if you use the **SHGetFileInfo** or **GetFileTitle** function. For more information about these functions, see the documentation included in the Win32 SDK.

Because the system uses three-letter extensions to describe a file type, do not use extensions to distinguish different forms of the same file type. For example, if your application has a function that automatically backs up a file, name the backup file Backup of *filename.ext* (using its existing extension) or some reasonable equivalent, not *filename*.bak. The latter implies a change of the file's type. Similarly, do not use a Windows filename extension unless your file fits the type description.

Long filenames can include any character, except the following.

\ / : * ? < > | “

When your application automatically supplies a filename, use a name that communicates information about its creation. For example, files created by a particular application should use either the application-supplied type name or the short type name as a proposed name — for example, worksheet or document. When that file exists already in the target directory, add a number to the end of the proposed name — for example, Document (2). When adding numbers to the end of a proposed filename, use the first number of an ordinal sequence that does not conflict with an existing name in that directory.

When saving a file, make certain you preserve the creation date of the file. For simple applications that open and save a file, this happens automatically. However, more sophisticated applications may create temporary files, delete the original file, and rename the temporary file to the original filename. In this case, the application needs to copy the creation date as well from the old file to the new, using the standard system functions. Certain system file management functionality may depend on the correct creation date.

When you create a filename, the system automatically creates an MS-DOS filename (alias) for a file. The system displays both the long filename and the MS-DOS filename in the property sheet for the file.

When a file is copied, use the words “Copy of” as part of the generated filename — for example, “Copy of Sample” for a file named “Sample.” If the prefix “Copy of” is already assigned to a file, include a number in parentheses — for example, “Copy (2) of Sample”. You can apply the same naming scheme to links, except the prefix is “Link to” or “Shortcut to.”

It is also important to support UNC paths for identifying the location of files and folders. UNC paths and filenames have the following form.

*Server**Share**Directory**Filename.ext*

Using UNC names enables the user to directly browse the network and open files without having to make explicit network connections.

Wherever possible, display the full name of a file (without the extension). The number of characters you'll be able to display depends somewhat on the font used and the context in which the name is displayed. In any case, supply enough characters such that the user can reasonably distinguish between names. Take into account common prefixes such as “Copy of” or “Shortcut to”. If you don't display the full name, indicate that it has been truncated by appending an ellipsis to the end of the name.

You can use an ellipsis to abbreviate path names, in a displayable, but noneditable situation. In this case, include at least the first two entries of the beginning and the end of the path, using ellipses as notation for the names in between, as in the following example.

\\My Server\My Share\...\My Folder\My File

When using an icon to represent a network resource, label the icon with the name of the resource. If you need to show the network context rather than using a UNC path, label the resource using the following format.

*Resource Name* on *Computer Name*

# Taskbar Integration

The system provides support for integrating your application’s interface with the taskbar. The following sections provide information on some of the capabilities and appropriate guidelines.

## Taskbar Window Buttons

When an application creates a primary window, the system automatically adds a taskbar button for that window and removes it when that window closes. For some specialized types of applications that run in the background, a primary window may not be necessary. In such cases, make certain you provide reasonable support for controlling the application using the commands available on the application's icon; it should not appear as an entry on the taskbar, however. Similarly, the secondary windows of an application should also not appear as a taskbar button.

The taskbar window buttons support drag and drop, but not in the conventional way. When the user drags an object over a taskbar window button, the system automatically restores the window. The user can then drop the object in the window.

## Status Notification

The system allows you to add status or notification information to the taskbar. Because the taskbar is a shared resource, add information to it that is of a global nature only or that needs monitoring by the user while working with other applications.

The **Shell_NotifyIcon** function provides support for adding a status item in the taskbar. For more information about this function, see the documentation included in the Win32 SDK.

Present status notification information in the form of a graphic supplied by your application, as shown in Figure 10.2.

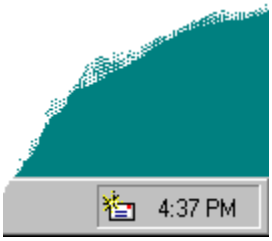



**Figure 10.2 Status indicator in the taskbar**

When adding a status indicator to the taskbar, also support the following interactions:

- Provide a pop-up window that displays further information or controls for the object represented by the status indicator when the user clicks with button 1. For example, the audio (speaker) status indicator displays a volume control. Use a pop-up window to supply for further information rather than a dialog box, because the user can dismiss the window by clicking elsewhere. Position the pop-up window near the status indicator so the user can navigate to it quickly and easily. Avoid displaying other types of secondary windows because they require explicit user interaction to dismiss them. If there is no information or control that applies, do not display anything.
- Display a pop-up menu for the object represented by the status indicator when the user clicks on the status indicator with button 2. On this menu, include commands that bring up property sheets or other windows related to the status indicator. For example, the audio status indicator provides commands that display the audio properties as well as the Volume Control mixer application.
- Carry out the default command defined in the pop-up menu for the status indicator when the user double-clicks.
- Display a tooltip that indicates what the status indicator represents. For example, this could include the name of the indicator, a value, or both.
- Provide the user an option to not display the status indicator, preferably in the property sheet for the object displaying the status indicator. This allows the user to determine which indicator to include in this shared space. You may need to provide an alternate means of conveying this status information when the user turns off the status indicator.

## Message Notification

When your application's window is inactive but must display a message, rather than displaying a message box on top of the currently active window and switching the input focus, flash your application's title bar and taskbar window button to notify the user of the pending message. This avoids interfering with the user's current activity but lets the user know a message is waiting. When the user activates your application's window, the application can display a message box.

The **FlashWindow** function supports flashing your title bar and taskbar window button. For more information about this function, see the documentation included in the Win32 SDK.

Use the system setting for the cursor blink rate for your flash rate. This allows the user to control the flash rate to a comfortable frequency.

The **GetCaretBlinkTime** function provides access to the current cursor blink rate setting. For more information about this function, see the documentation included in the Win32 SDK.

Rather than flashing the button continually, you can flash the window button only a limited number of times (for example, three), then leave the button in the highlighted state, as shown in Figure 10.3. This lets the user know there is still a pending message.


**Figure 10.3 Flashing a taskbar button to notify a user of a pending message**

This cooperative means of notification is preferable unless a message relates to the system integrity of the user's data, in which case your application may immediately display a system modal message box. In such cases, flush the input queue so that the user does not inadvertently select a choice in that message box.

# Application Desktop Toolbars

The system supports applications supplying their own desktop toolbars, also referred to as access bars or appbars, that operate similarly to the Windows taskbar. These may be docked to the edges of a screen and provide access to controls, such as buttons, for specific functions.

The system supports the same auto-hide behavior for application desktop toolbars as it does for the taskbar. This allows the desktop toolbar to only be visible when the user moves the pointer to the edge of the screen. The system also provides the “always on top” behavior used by the taskbar. When the user sets this property, the taskbar always appears on top (in the Z order) of any windows and also acts as a boundary for windows set to maximize to the display screen size.

Desktop toolbars can also be undocked and displayed as a palette window or redocked at a different edge of the screen. In the undocked, displayed as a palette window state, the toolbar no longer constrains other windows. However, if it supports the Always on Top property, it remains on top of other application windows.

For more information on the recommended behavior for undocking and redocking toolbars, see Menus, Controls, and Toolbars.

Before designing a desktop toolbar, consider whether your application’s tasks really require one. Remember that a desktop toolbar will potentially affect the visible area for all applications. Only provide one for frequently used interfaces that can be applied across applications and always design it to be an optional interface, allowing the user to close it or otherwise configure it not to appear. You may also want to consider removing it when a specific application or applications are closed.

When creating your own desktop toolbar, model its behavior on the taskbar. Consider using the system’s notification of when the taskbar’s auto-hide or Always on Top property changes to apply a desktop toolbar you provide. If this does not fit your design, be certain to provide your own property sheet for setting these attributes for your desktop toolbar. Note that the system only supports auto-hide functionality for one desktop toolbar on each edge of the display. In addition, always provide a pop-up menu to access commands that apply to your desktop toolbar, such as Close, Move, Size, and Properties (but not the commands included on the desktop toolbar).

You can choose to display a desktop toolbar when the user runs a specific application, or by creating a separate application and including a shortcut icon to it in the system’s Startup folder. Preferably set the initial size and position of your desktop toolbar so that it does not interfere with other desktop toolbars or the taskbar. However, the system does support multiple desktop toolbars to be docked along the same edge of the display screen. When docking on the same edge as the taskbar, the system places the taskbar on the outermost edge.

Your desktop toolbar can include any type of control. A desktop toolbar can also be a drag and drop target. Follow the recommendations outlined in this guide for supporting appropriate interaction.

## Full-Screen Display

Although the taskbar and application desktop toolbars normally constrain or clip windows displayed on the screen, you can define a window to the full extent of the display screen. Because this is not the typical form of interaction, only consider using full-screen display for very special circumstances, such as a slide presentation, and only when the user explicitly chooses a command for this purpose. Make certain you provide an easy way for the user to return to normal display viewing. For example, you can display an on-screen button when the user moves the pointer that restores the display when the user clicks it. In addition, keyboard interfaces, like ALT+TAB and ESC, should automatically restore the display.

Remember that desktop toolbars, including the taskbar, should support auto-hide options that allow the user to configure them to reduce their visual impact on the screen. Consider whether this auto-hide capability may be sufficient before designing your application to require a full-screen presentation. Advising the user to close or hide desktop toolbars may provide you with sufficient space without having to use the full display screen.

# Recycle Bin Integration

The Recycle Bin provides a repository for deleted files. If your application includes a facility for deleting files, support the Recycle Bin interface. You can also support deletion to the Recycle Bin for nonfile objects by first formatting the deleted data as a file by writing it to a temporary file and then calling the system functions that support the Recycle Bin.

The **SHFileOperation** function supports deletion using the Recycle Bin interface. For more information about this function, see the documentation included in the Win32 SDK.

## Control Panel Integration

The Windows Control Panel includes special objects that let users configure aspects of the system. Your application can add Control Panel objects or add property pages to the property sheets of existing Control Panel objects.

## Adding Control Panel Objects

You can create your own Control Panel objects. Most Control Panel objects supply only a single secondary window, typically a property sheet. Define your Control Panel object to represent a concrete object rather than an abstract idea.

Every Control Panel object is a dynamic-link library. To ensure that the dynamic-link library can be automatically loaded by the system, set the file's extension to .CPL and install it in the Windows System directory.

The system automatically caches information about Control Panel objects in order to provide quick user access, provided that the Control Panel object supports the correct system interfaces. For more information about developing Control Panel objects, see the documentation included in the Win32 SDK.

## Adding to the Passwords Object

The Passwords object in Control Panel supplies a property sheet that allows the user to set security options and manage passwords for all password-protected services in the system. The Passwords object also allows you to add the name of a password-protected service to the object's list of services and use the Windows login password for all password-protected services in the system.

When you add your service to the Passwords object, the name of the service appears in the Select Password dialog box that appears when the user chooses Change Other Passwords. The user can then change the password for the service by selecting the name and filling in the resulting dialog box. The name of your service also appears in the Change Windows Password dialog box; the name appears with a check box next to it. By setting the check box option, the user chooses to keep the password for the service identical to the Windows login password. Similarly, the user can disassociate the service from the Windows login password by toggling the check box setting off.

To add your service to the Passwords object, register your service under the **HKEY_LOCAL_MACHINE** key.

For more information about registering your password service, see the documentation included in the Win32 SDK.

**HKEY_LOCAL_MACHINE**
**System**
**CurrentControlSet**
**Control**
**PwdProvider**
***Provider Name*** *Value Name = Value*

You can also add a page to the property sheet of the Passwords object to support other security-related services that the user can set as property values. Add a property page if your application provides security-related functionality beyond simple activation and changing of passwords. To add a property page, follow the conventions for adding shell extensions.

## Plug and Play Support

Plug and Play is a feature of Windows that, with little or no user intervention, automatically installs and configures drivers when their corresponding hardware peripherals are plugged into a PC. This feature applies to peripherals designed according to the Plug and Play specification. Supporting and appropriately adapting to Plug and Play hardware change can make your application easier to use. Following are some examples of supporting Plug and Play:

- Resizing your windows and toolbars relevant to screen size changes.
- Prompting users to shut down and save their data when the system issues a low power warning.
- Warning users about open network files when undocking their computers.
- Saving and closing files appropriately when users eject or remove removable media or storage devices or when network connections are broken.

## System Settings and Notification

The system provides standard metrics and settings for user interface aspects, such as colors, fonts, border width, and drag rectangle (used to detect the start of a drag operation). The system also notifies running applications when its settings change. When your application starts up, query the system to set your application's user interface to match the system parameters to ensure visual and operational consistency. Also, design your application to adjust itself appropriately when the system notifies it of changes to these settings.

The **GetSystemMetrics**, **Get-SysColor**, and **SystemParametersInfo** functions and the WM_SETTINGSCHANGE message are important to consider when supporting standard system settings. For more information about these system interfaces, see the documentation included in the Win32 SDK.

## Modeless Interaction

When designing your application, try to ensure that it is as interactive and nonmodal as possible. Here are some suggested ways of doing this:

- Use modeless secondary windows wherever possible.
- Segment processes, like printing, so you do not need to load the entire application to perform the operation.
- Make long processes run in the background, keeping the foreground interactive. For example, when something is printing, it should be possible to minimize the window even if the document cannot be altered. The multitasking support of Windows provides for defining separate processes, or *threads*, in the background.

For more information about threads, see the documentation included in the Win32 SDK.

# Introduction

Microsoft OLE provides a set of system interfaces that enables users to combine objects supported by different applications. This topic outlines guidelines for the interface for OLE embedded and OLE linked objects; you can apply many of these guidelines to any implementation of containers and their components.

 Undo Operations for Active and Open Objects

 Displaying Messages

 Object Application Messages

 OLE Linked Object Messages

 Status Line Messages

## The Interaction Model

As data becomes the major focus of interface design, its content is what occupies the user's attention, not the application managing it. In such a design, data is not limited to its native creation and editing environment; that is, the user is not limited to creating or editing data only within its associated application window. Instead, data can be transferred to other types of containers while maintaining its viewing and editing capability in the new container. Compound documents are a common example and illustration of the interaction between containers and their components, but they are not the only expression of this kind of object relationship that OLE can support.

Figure 11.1 shows an example of a compound document. The document includes word-processing text, tabular data from a spreadsheet, a sound recording, and pictures created in other applications.

# Classical CD Review

**by Thomas D. Becker**

The introduction of the compact disc has had a far greater impact on the recording industry than anyone could have imagined, especially the manufacturers of vinyl long play (LP) albums. With the 1991 sales totals in, compact disc is clearly the preferred recording medium for American ears. In addition to audio compact discs, CD-ROMs are appearing on the market offering a multimedia experience of the classical repetoire. The Microsoft Composer Collection brings you the ability to enter the lives and minds of three astounding musical geniuses. That's because the Composer Collection contains three CD-ROM titles full of music, information, and entertainment. They are: Microsoft Multimedia Mozart, Microsoft Multimedia Stravinsky, and Microsoft Multimedia Beethoven. These works are reviewed below - be sure to check them out! -*TDB*

**U.S. Compact Disc vs. LP Sales ($)**

| | 1983 | 1987 | 1991 |
|---|---|---|---|
| **CDs** | 6,345K | 18,652K | 32,657K |
| **LPs** | 31,538K | 26,571K | 17,429K |
| **Total** | 37,883K | 45,223K | 50,086K |

---


Multimedia Mozart: The Dissonant Quartet

The Voyager Company
Microsoft

In the words of author and music scholar Robert Winter, the string quartet in the eighteenth century was regarded as one of the "most sublime forms of communication." The String Quartet in C Major is no exception. Discover the power and the beauty of this music with Microsoft Multimedia Mozart: *The Dissonant Quartet*, and enter the world in which Mozart created his most memorable masterpieces. Sit back and enjoy *The Dissonant Quartet* in its entirety, or browse around, exploring its themes and emotional dynamics in depth. View the entire piece in a single-screen overview with the *Pocket Audio Guide*.


Multimedia Stravinsky: The Rite of Spring

The Voyager Company
Microsoft

Multimedia Stravinsky: *The Rite of Spring* offers you an in-depth look at this controversial composition. Author Robert Winter provides a fascinating commentary that follows the music, giving you greater understanding of the subtle dynamics of the instruments and powerful techniques of Stravinsky. You'll also have the opportunity to discover the ballet that accompanied *The Rite of Spring* in performance. Choreographed by Sergei Diaghilev, the ballet was as unusual for its time as the music. To whet your appetite, play this audio clip.


Multimedia Beethoven: The Ninth Symphony

The Voyager Company
Microsoft

Multimedia Beethoven: The Ninth Symphony is one of a series of engaging, informative, and interactive musical explorations from Microsoft. It enables you to examine Beethoven's world and life, and explore the form and beauty of one of his foremost compositions. You can compare musical themes, hear selected orchestral instruments, and see the symphonic score come alive. Multimedia Beethoven: *The Ninth Symphony* is an extraordinary opportunity to learn while you listen to one of the world's musical treasures. Explore this inspiring work at your own pace in *A Close Reading*. As you listen to a superb performance of Beethoven's



**Figure 11.1 A compound document**

How was this music review created? First, a user created a document and typed the text, then moved, copied, or linked content from other documents. Data objects that, when moved or copied, retain their native, full-featured editing and operating capabilities in their new container are called *OLE embedded objects*.

A user can also link information. An *OLE linked object* represents or provides access to another object that is in another location in the same container or in a different, separate container.

Generally, containers support any level of nested OLE embedded and linked objects. For example, a user can embed a chart in a worksheet, which, in turn, can be embedded in a word-processing document. The model for interaction is consistent at each level of nesting.

## Creating OLE Embedded and OLE Linked Objects

OLE embedded and linked objects are the result of transferring existing objects or creating new objects of a particular type.

## Transferring Objects

Transferring objects into a document follows basic command and direct manipulation interaction methods. The following sections provide additional guidelines for these commands when you use them to create OLE embedded or linked objects.

For more information about command transfer and direct manipulation transfer methods, see General Interaction Techniques.

## The Paste Command

As a general rule, using the Paste command should result in the most complete representation of a transferred object; that is, the object is embedded. However, containers that directly handle the transferred object can accept it optionally as native data instead of embedding it as a separate object, or as a partial or transformed form of the object if that is more appropriate for the destination container.

Use the format of the Paste command to indicate to the user how a transferred object is incorporated by a container. When the user copies a file object, if the container can embed the object, include the object's filename as a suffix to the Paste command. If the object is only a portion of a file, use the short type name — for example, Paste Worksheet or Paste Recording — as shown in Figure 11.2. A short type name can be derived from information stored in the registry. A Paste command with no name implies that the data will be pasted as native information.

For more information about type names and the system registry, see Integrating with the System, and the OLE documentation included in the Microsoft Win32 Software Development Kit (SDK).


**Figure 11.2 The Paste command with short type name**

## The Paste Special Command

Supply the Paste Special command to give the user explicit control over pasting in the data as native information, an OLE embedded object, or an OLE linked object. The Paste Special command displays its associated dialog box, as shown in Figure 11.3. This dialog box includes a list box with the possible formats that the data can assume in the destination container.

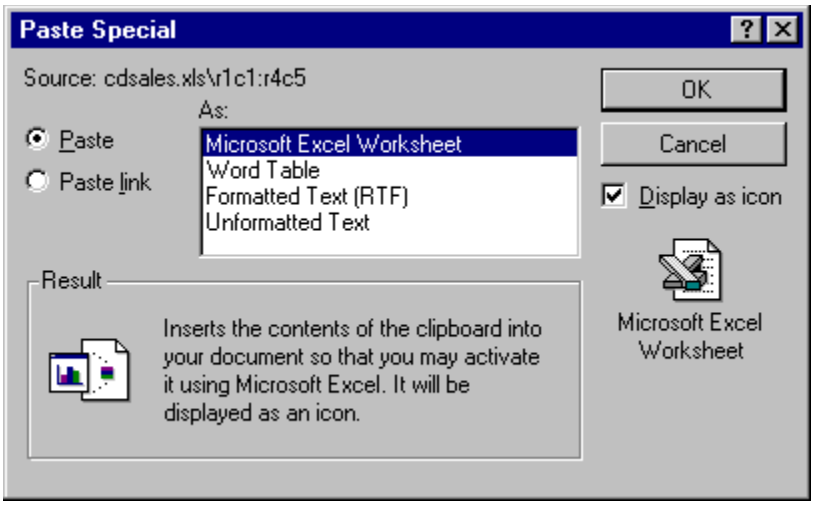



**Figure 11.3 The Paste Special dialog box**

The Win32 SDK includes the Paste Special dialog box and other OLE-related dialog boxes that are described in this topic.

In the formats listed in the Paste Special dialog box, include the object's full type name first, followed by other appropriate native data forms. When a linked object has been cut or copied, precede its object type by the word "Linked" in the format list. For example, if the user copies a linked Microsoft Excel worksheet, the Paste Special dialog box shows "Linked Microsoft Excel Worksheet" in the list of format options because it inserts an exact duplicate of the original linked worksheet. Native data formats begin with the destination application's name and can be expressed in the same terms the destination identifies in its own menus. The initially selected format in the list corresponds to the format that the Paste command uses. For example, if the Paste command is displayed as Paste *Object Filename* or Paste *Short Type Name* because the data to be embedded is a file or portion of a file, this is the format that is initially selected in the Paste Special list box.

To support creation of a linked object, the Paste Special dialog box includes a Paste Link option. Figure 11.4 shows this option.

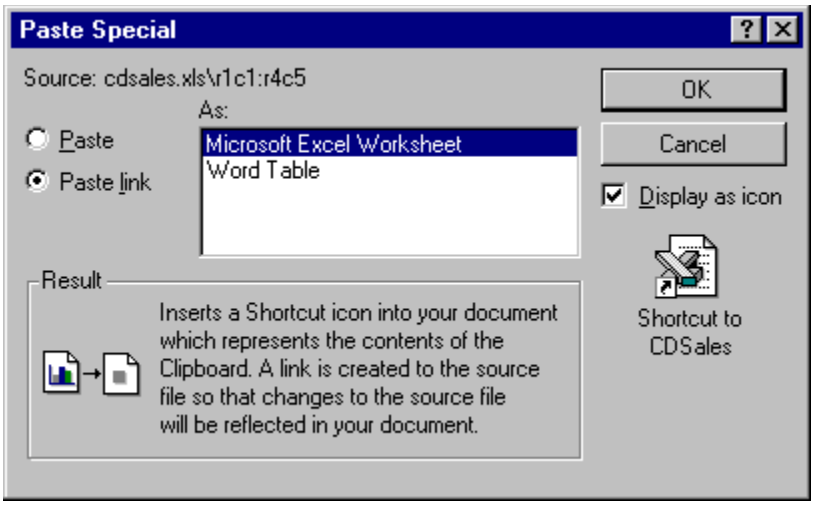



**Figure 11.4 Paste Special dialog box with Paste Link option set**

A Display As Icon check box allows the user to choose displaying the OLE embedded or linked object as an icon. At the bottom of the dialog box is a section that includes text and pictures that describe the result of the operation. Table 11.1 lists the descriptive text for use in the Paste Special dialog box.

**Table 11.1 Descriptive Text for Paste Special Command**

| Function | Resulting text |
|---|---|
| Paste as an OLE embedded object. | "Inserts the contents of the Clipboard into your document so you that you may activate it using *CompanyName ApplicationName*." |
| Paste as an OLE embedded object so that it appears as an icon. | "Inserts the contents of the Clipboard into your document so you that you may activate it using *CompanyName ApplicationName* application. It will be displayed as an icon." |
| Paste as native data. | "Inserts the contents of the Clipboard into your document as *Native Type Name*. [*Optional additional Help sentence*.]" |
| Paste as an OLE linked object. | "Inserts a picture of the contents of the Clipboard into your document. Paste Link creates a link to the source file so that changes to the source file will be reflected in your document." |
| Paste as an OLE linked object so that it appears as a shortcut icon. | "Inserts a Shortcut icon into your document which represents the contents of the Clipboard. A link is created to the source file so that changes to the source file will be reflected in your document." |
| Paste as linked native data. | "Inserts the contents of the Clipboard into your document as *Native Type Name*. A link is created to the source file so that changes to the source file |

will be reflected in your document.”

## The Paste Link, Paste Shortcut, and Create Shortcut Commands

If linking is a common function in your application, you can optionally include a command that optimizes this process. Use a Paste Link command to support creating a linked object or linked native data. When using the command to create a linked object, include the name of the object preceded by the word “to” — for example, “Paste Link to Latest Sales.” Omitting the name implies that the operation results in linked native data.

Use a Paste Shortcut command to support creation of a linked object that appears as a shortcut icon. You can also include a Create Shortcut command that creates a shortcut icon in the container. Apply these commands to containers where icons are commonly used.

## Direct Manipulation

You should also support direct manipulation interaction techniques, such as drag and drop, for creating OLE embedded or linked objects. When the user drags a selection into a container, the container application interprets the operation using information supplied by the source, such as the selection's type and format, and by the destination container's own context, such as the container's type and its default transfer operation. For example, dragging a spreadsheet cell selection into a word-processing document can result in an OLE embedded table object. Dragging the same cell selection within the spreadsheet, however, would likely result in simply transferring the data in the cells. Similarly, the destination container in which the user drops the selection can also determine whether the dragging operation results in an OLE linked object.

For more information about using direct manipulation for moving, copying, and linking objects, see General Interaction Techniques.

For nondefault OLE drag and drop, the container application displays a pop-up menu with appropriate transfer commands at the end of the drag. The choices may include multiple commands that transfer the data in a different format or presentation. For example, as shown in Figure 11.5, a container application could offer the following choices for creating links: Link Here, Link *Short Type Name* Here, and Create Shortcut Here, respectively resulting in a native data link, an OLE linked object displayed as content, and an OLE linked object displayed as an icon. The choices depends on what the container can support.

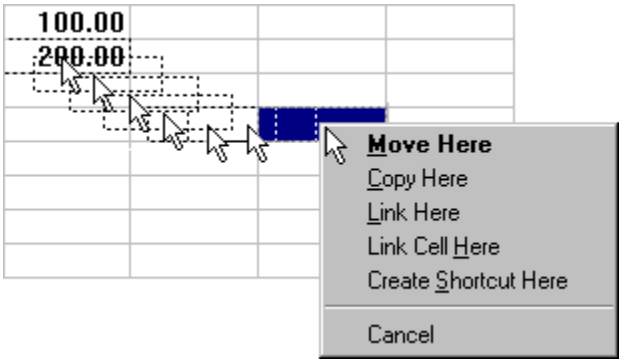



**Figure 11.5 Containers can offer different OLE link options**

The default appearance of a transferred object also depends on the destination container application. For most types of documents, make the default command one that results in the data or content presentation of the object (or in the case of an OLE linked object, a representation of the content), rather than as an icon. If the user chooses Create Shortcut Here as the transfer operation, display the transferred object as an icon. If the object cannot be displayed as content — for example, because it does not support OLE — always display the object as an icon.

## Transfer of Data to Desktop

The system allows the user to transfer data selection within a file to the desktop or folders providing that the application supports the OLE transfer protocol. For move or copy operations — using the Cut, Copy, and Paste commands or direct manipulation — the transfer operation results in a file icon called a *scrap*. A link operation also results in a shortcut icon that represents a shortcut into a document.

When the user transfers a scrap into a container supported by your application, integrate it as if it were being transferred from its original source. For example, if the user transfers a selected range of cells from a spreadsheet to the desktop, it becomes a scrap. If the user transfers the resulting scrap into a word-processing document, the document should incorporate the scrap as if the cells were transferred directly from the spreadsheet. Similarly, if the user transfers the scrap back into the spreadsheet, the spreadsheet should integrate the cells as if they had been transferred within that spreadsheet. (Typically, internal transfers of native data within a container result in repositioning the data rather than transforming it.)

## Inserting New Objects

In addition to transferring objects, you can support user creation of OLE embedded or linked objects by generating a new object based on an existing object or object type and inserting the new object into the target container.

## The Insert Object Command

Include an Insert Object command on the menu responsible for creating or importing new objects into a container, such as an Insert menu. If no such menu exists, use the Edit menu. When the user selects this command, display the Insert Object dialog box, as shown in Figure 11.6. This dialog box allows the user to generate new objects based on their object type or an existing file.

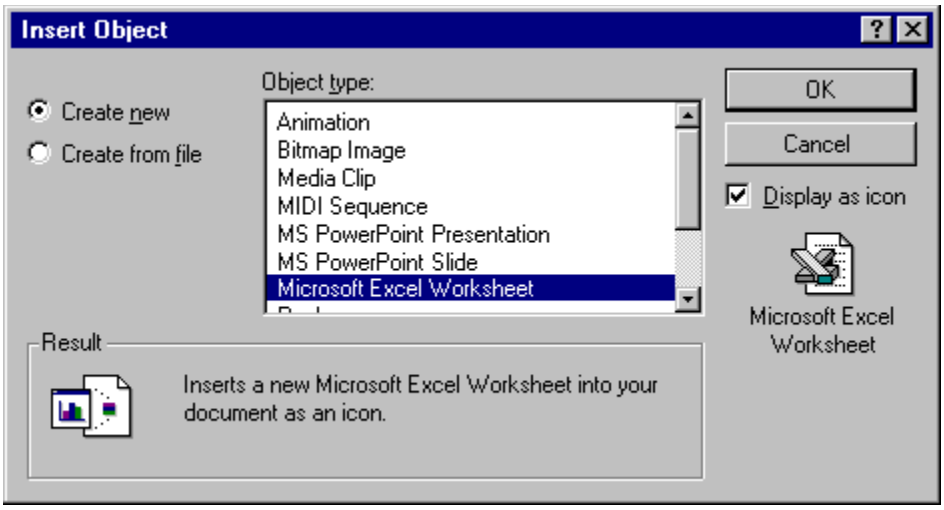



**Figure 11.6 The Insert Object dialog box**

The type list is composed of the type names of registered types. When the user selects a type from the list box and chooses the OK button, an object of the selected type is created and embedded.

For more information about type names and the registry, see Integrating with the System.

The user can also create an OLE embedded or linked object from an existing file, using the Create From File and Link options. When the user sets these options and chooses the OK button, the result is the same as directly copying or linking the selected file.

When the user chooses the Create From File option button, the Object Type list is removed, and a text box and Browse button appear in its place, as shown in Figure 11.7. Ignore any selection formerly displayed in the Object Type list box (shown in Figure 11.6).

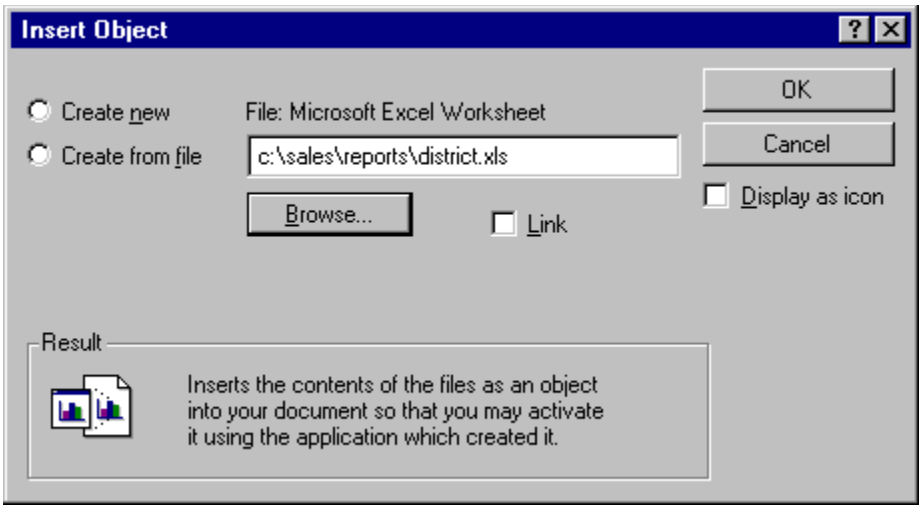



**Figure 11.7 Creating an OLE embedded object from an existing file**

The text box initially includes the current directory as the selection. The user can edit the current directory path when specifying a file. As an alternative, the Browse button displays an Open dialog box that allows the user to navigate through the file system to select a file. Use the file's type to determine the type of the resulting OLE object.

Use the Link check box to support the creation of an OLE linked object to the file specified. The Insert Object dialog box displays this option only when the user chooses the Create From File option. This means that a user cannot insert an OLE linked object when choosing the Create New option button, because linked objects can be created only from existing files.

The Display As Icon check box in the Insert Object dialog box enables the user to specify whether to display the OLE object as an icon. When this option is set, the icon appears beneath the check box. An OLE linked object displayed as an icon is the equivalent of a shortcut icon. It should appear with the link symbol over the icon.

If the user chooses a non-OLE file for insertion, it can be inserted only as an icon. The result is an OLE package. A *package* is an OLE encapsulation of a file so that it can be embedded in an OLE container. Because packages support limited editing and viewing capabilities, support OLE for all your object types so they will not be converted into packages.

At the bottom of the Insert Object dialog box, text and pictures describe the final outcome of the insertion. Table 11.2 outlines the syntax of descriptive text to use within the Insert Object dialog box.

**Table 11.2 Descriptive Text for Insert Object Dialog Box**

| Function | Resulting text |
|---|---|
| Create a new OLE embedded object based on the selected type. | "Inserts a new *Type Name* into your document." |
| Create a new OLE embedded object based on the selected type and display it as an icon. | "Inserts a new *Type Name* into your document as an icon." |
| Create a new OLE embedded object based on a selected file. | "Inserts the contents of the file as an object into your document so that you may activate it using the application which created it." |
| Create a new OLE embedded object based on a selected file (copies the file) and display it as an icon. | "Inserts the contents of the file as an object into your document so that you may activate it using the application |

| | |
|---|---|
| | which created it. It will be displayed as an icon." |
| Create an OLE linked object that is linked to a selected file. | "Inserts a picture of the file contents into your document. The picture will be linked to the file so that changes to the file will be reflected in your document." |
| Create an OLE linked object that is linked to a selected file and display it as a Shortcut icon. | "Inserts a Shortcut icon into your document which represents the file. The Shortcut icon will be linked to the original file, so that you can quickly open the original from inside your document." |

You can also use the context of the current selection in the container to determine the format of the newly created object and the effect of it being inserted into the container. For example, an inserted graph can automatically reflect the data in a selected table. Use the following guidelines to support predictable insertion:

- If an inserted object is not based on the current selection, follow the same conventions as for a Paste command and add or replace the selection depending on the context. For example, in text or list contexts, where the selection represents a specific insertion location, replace the active selection. For nonordered or Z-ordered contexts, where the selection does not represent an explicit insertion point, add the object, using the destination's context to determine where to place the object.

For more information about the guidelines for inserting an object with a Paste command, see General Interaction Techniques.

- If the new object is automatically connected (linked) to the selection (for example, an inserted graph based on selected table data), insert the new object in addition to the selection and make the inserted object the new selection.

After inserting an OLE embedded object, activate it for editing. However, if the user inserts an OLE linked object, do not activate the object.

## Other Techniques for Inserting Objects

The Insert Object command provides support for inserting all registered OLE objects. You can include additional commands tailored to provide access to common or frequently used object types. You can implement these as additional menu commands or as toolbar buttons or other controls. These buttons provide the same functionality as the Insert Object dialog box, but perform more efficiently. Figure 11.8 illustrates two examples. The drawing button inserts a new blank drawing object; the graph button creates a new graph that uses the data values from a currently selected table.

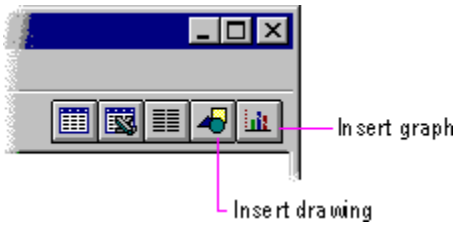



**Figure 11.8 Using toolbar buttons for creating new objects**

## Displaying Objects

While a container can control whether to display an OLE embedded or linked object in its content or icon presentation, the container requests the object to display itself. In the content presentation, the object may be visually indistinguishable from native objects, as shown in Figure 11.9.

The **GetSysColor** function provides the current settings for window text color (COLOR_WINDOWTEXT) and grayed text color (COLOR_GRAYTEXT). For more information about this function, see the documentation included in the Win32 SDK.

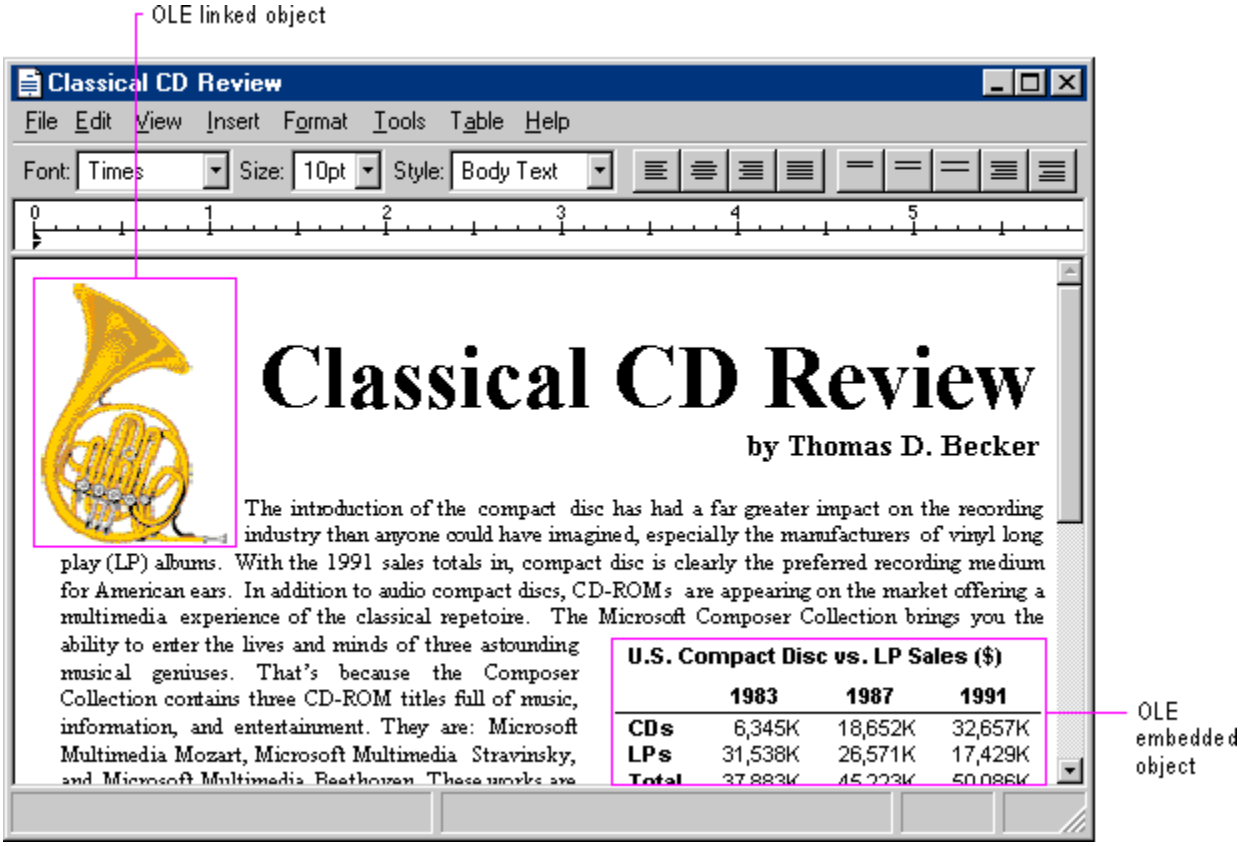



**Figure 11.9 A compound document containing OLE objects**

You may find it preferable to enable the user to visually identify OLE embedded or linked objects without interacting with them. To do so, you can include a Show Objects command that, when chosen, displays a solid border, one pixel wide, drawn in the window text color around the extent of an OLE embedded object and a dotted border around OLE linked objects (shown in Figure 11.10). If the container application cannot guarantee that an OLE linked object is up-to-date with its source because of an unsuccessful automatic update or a manual link, the system should draw a dotted border using the system grayed text color to suggest that the OLE linked object is out of date. The border should be drawn around a container's first-level objects only, not objects nested below this level.

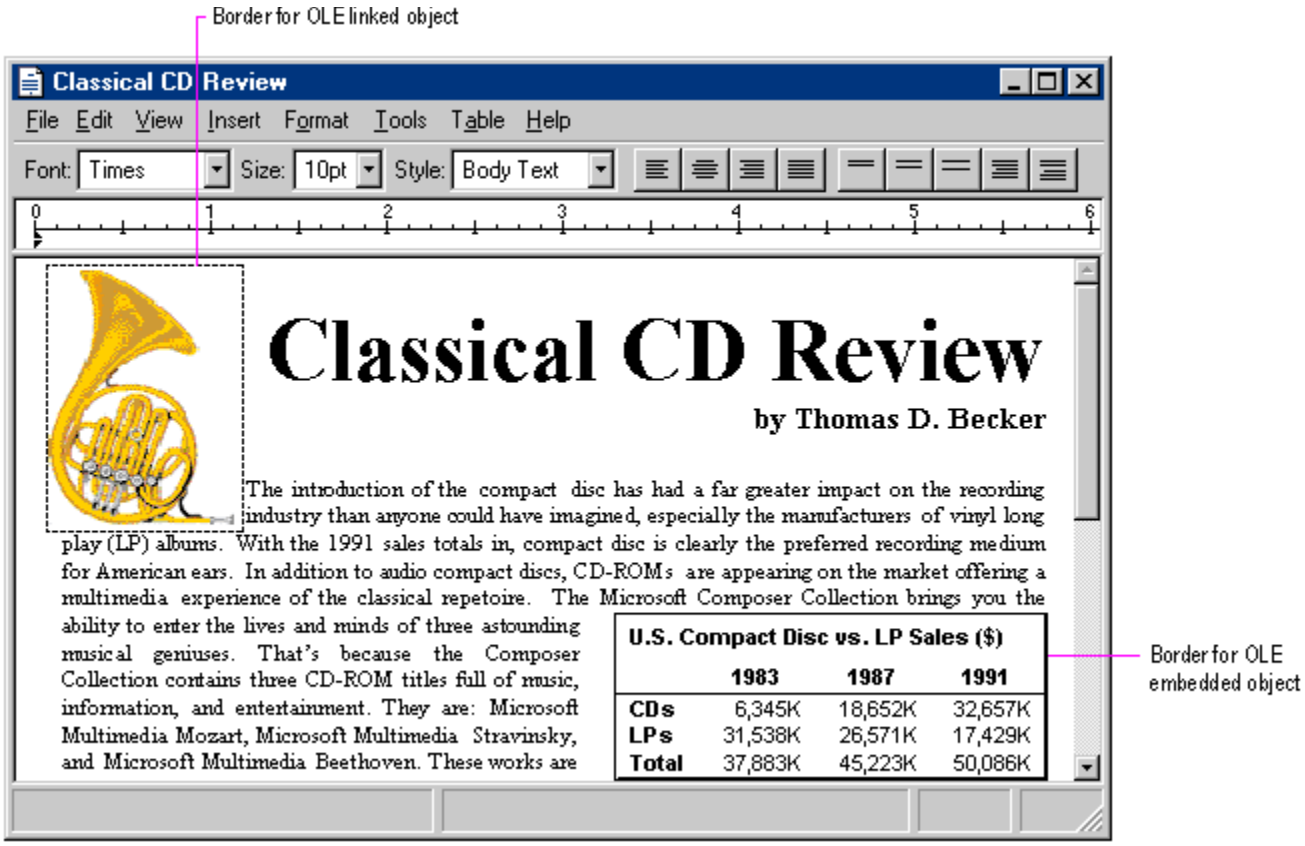



**Figure 11.10 Identifying OLE objects using borders**

If these border conventions are not adequate to distinguish OLE embedded and linked objects, you can optionally include additional distinctions; however, make them clearly distinct from the appearance for any standard visual states and distinguish OLE embedded from OLE linked objects.

Whenever the user creates an OLE linked or embedded object with the Display As Icon check box set, display the icon using the icon of its type, unless the user explicitly changes it. A linked icon also includes the shortcut graphic. If an icon is not registered in the registry for the object, use the system-generated icon.

An icon includes a label. When the user creates an OLE embedded object, define the icon's label to be one of the following, based on availability:

- The name of the object, if the object has an existing human-readable name such as a filename without its extension.
- The object's registered short type name (for example, Picture, Worksheet, and so on), if the object does not have a name.
- The object's registered full type name (for example, a bitmap image, a Microsoft Excel Worksheet), if the object has no name or registered short type name.
- "Document" if an object has no name, a short type name, or a registered type name.

When an OLE linked object is displayed as an icon, define the label using the source filename as it appears in the file system, preceded by the words "Shortcut to" — for example, "Shortcut to Annual Report." The path of the source is not included. Avoid displaying the filename extension unless the user chooses the system option to display extensions or the file type is not registered.

The system provides support to automatically format the name correctly if you use the **GetIconOfFile** function. For more information about this function, see the OLE documentation included in the Win32

SDK.

When the user creates an OLE object linked to only a portion of a document (file), follow the same conventions for labeling the shortcut icon. However, because a container can include multiple links to different portions of the same file, you may want to provide further identification to differentiate linked objects. You can do this by appending a portion of the end of the link path (moniker). For example, you may want to include everything from the end of the path up to the last or next to last occurrence of a link path delimiter. OLE applications should use the exclamation point (!) character for identifying a data range. However, the link path may include other types of delimiters. Be careful when deriving an identifier from the link path to format the additional information using only valid filename characters so that if the user transfers the shortcut icon to a folder or the desktop, the name can still be used.

## Selecting Objects

An OLE embedded or linked object follows the selection behavior and appearance techniques supported by its container; the container application supplies the specific appearance of the object. For example, Figure 11.11 shows how the linked drawing of a horn is handled as part of a contiguous selection in the document.

For information about selection, see General Interaction Techniques. For information about selection appearance, see Visual Design.

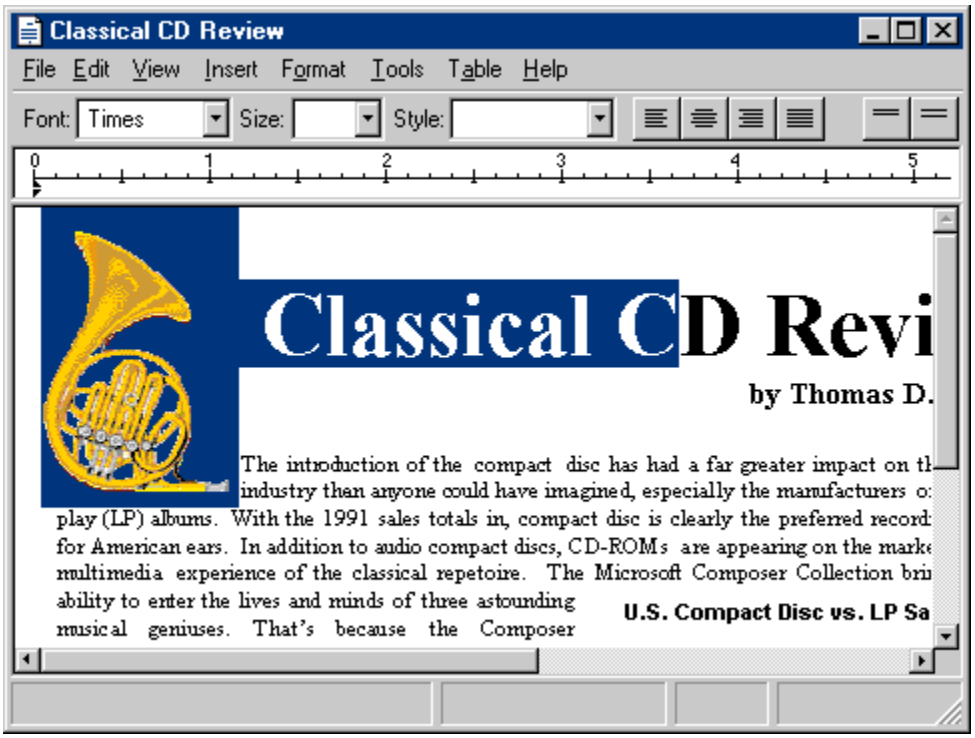



**Figure 11.11 An OLE linked object as part of a multiple selection**

When the user individually selects the object, display the object with an appropriate selection appearance. For example, for the content view of an object, display it with handles, as shown in Figure 11.12. For OLE linked objects, overlay the content view's lower left corner with the shortcut graphic. In addition, if your application's window includes a status bar that displays messages, display an appropriate description of how to activate the object (see Table 11.3).

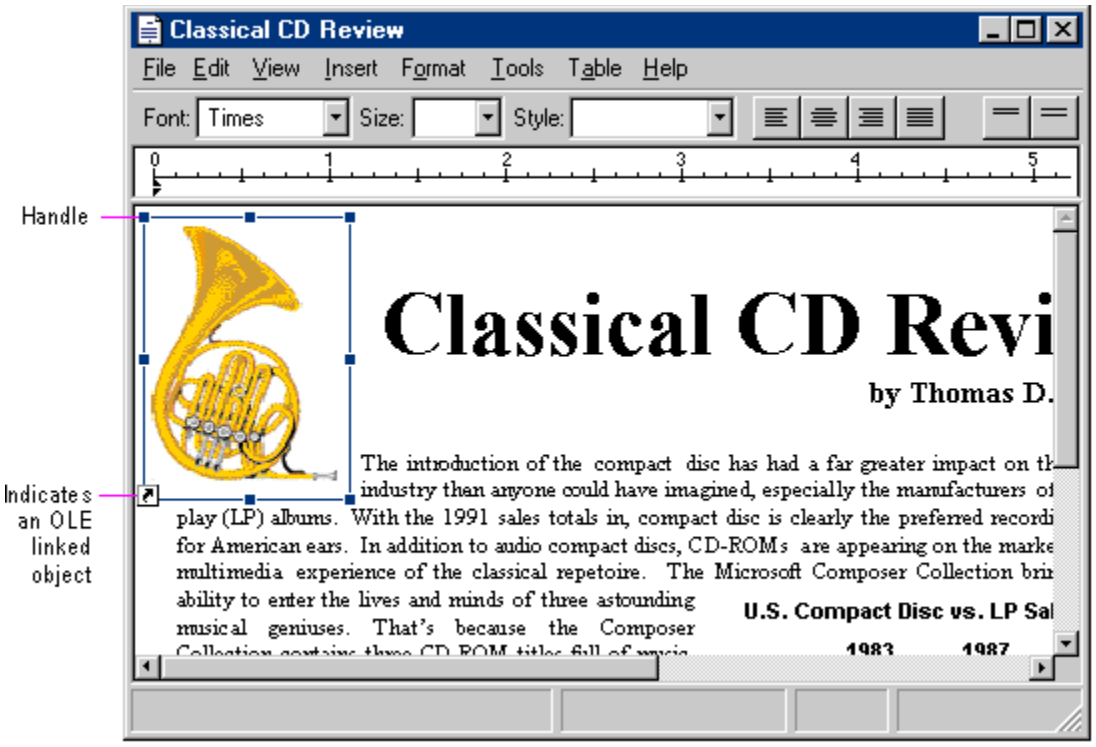



**Figure 11.12 An individually selected OLE linked object**

When the object is displayed as an icon, use the same selection appearance as for selected icons in folders and on the desktop, as shown in Figure 11.13.

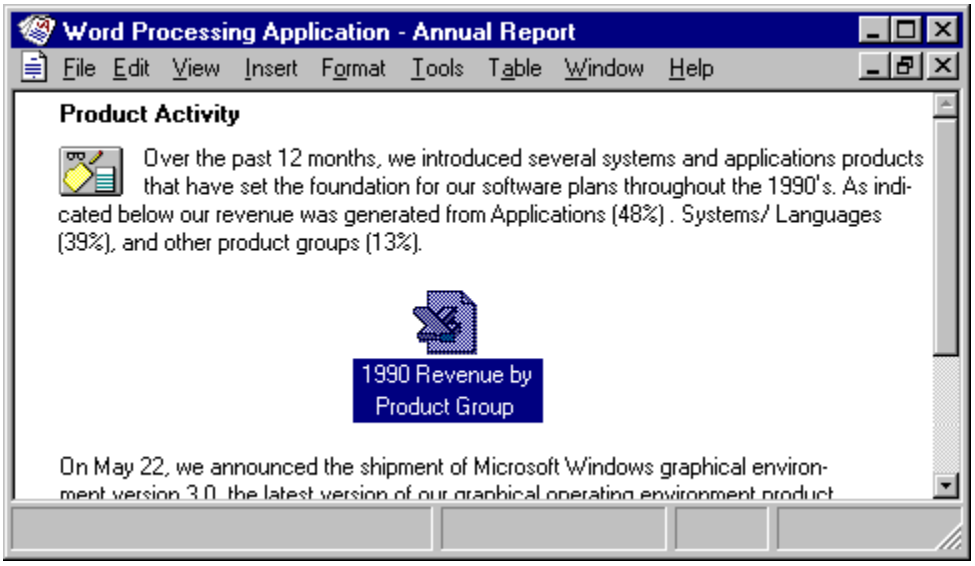



**Figure 11.13 A selected OLE object displayed as an icon**

## Accessing Commands for Selected Objects

A container application always displays the commands that can be applied to its objects. When the user selects an OLE embedded or linked object as part of the selection of native data in a container, enable commands that apply to the selection as a whole. When the user individually selects the object, enable only commands that apply specifically to the object. The container application retrieves these commands from what has been registered by the object's type in the registry and displays these commands in the menus that are supplied for the object. If your application includes a menu bar, include the selected object's commands on a submenu of the Edit menu, or as a separate menu on the menu bar. Use the name of the object as the text for the menu item. If you use the short type name as the name of the object, add the word "Object." For an OLE linked object, use the short type name, preceded by the word "Linked." Figure 11.14 shows these variations.

You can also support opera-tions based on the selection appearance. For example, you can support operations, such as resizing, using the handles you supply. When the user resizes a selected OLE object, however, scale the presentation of the object, because there is no method by which another operation, such as cropping, can be applied to the OLE object.

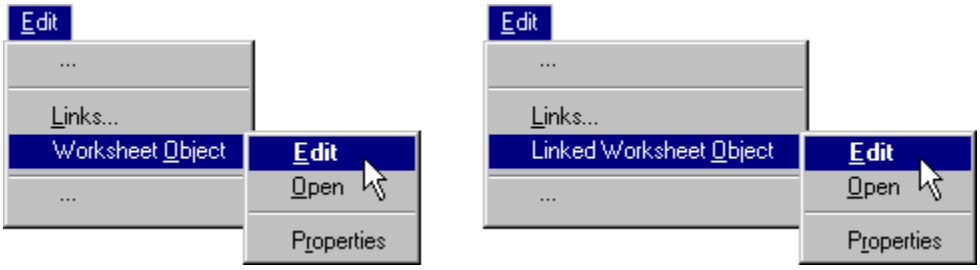



**Figure 11.14 Drop-down menus for selected OLE object**

Define the first letter of the word "Object", or its localized equivalent, as the access character for keyboard users. When no object is selected, display the command with just the text, "Object", and disable it.

A container application should also provide a pop-up menu for a selected OLE object (shown in Figure 11.15), displayed using the standard interaction techniques for pop-up menus (clicking with mouse button 2). Include on this menu the commands that apply to the object as a whole as a unit of content, such as transfer commands and the object's registered commands. In the pop-up menu, display the object's registered commands as individual menu items rather than in a cascading menu. It is not necessary to include the object's name or the word "Object" as part of the menu item text.

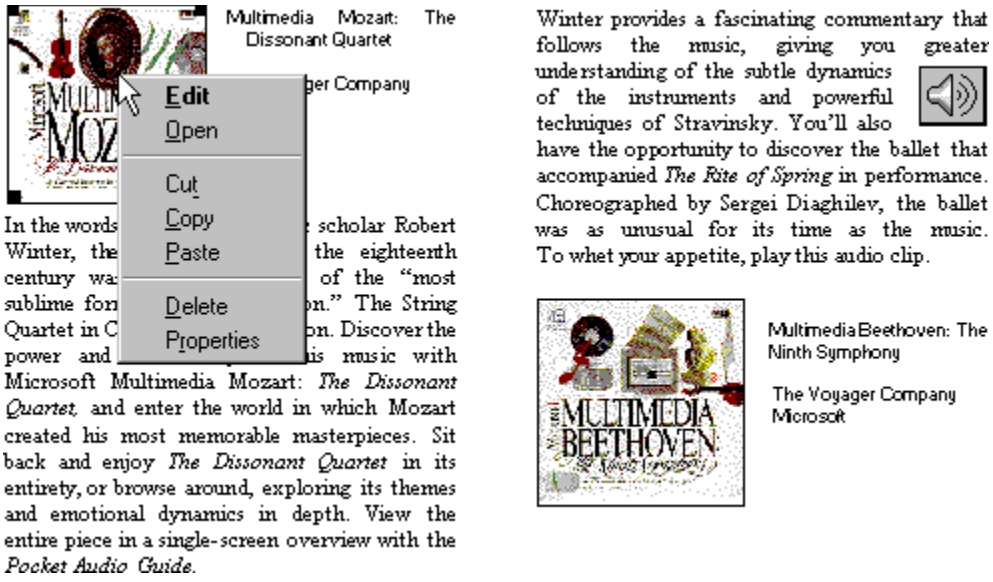



**Figure 11.15 Pop-up menu for an OLE embedded picture**

In the drop-down menu and the pop-up menu, include a Properties command. You can also include commands that depend on the state of the object. For example, a media object that uses Play and Rewind as operations disables Rewind when the object is at the beginning of the media object.

If an object's type is not registered, you still supply any commands that can be appropriately applied to the object as content, such as transfer commands, alignment commands, and an Edit and Properties command. When the user chooses the Edit command, display the system-supplied message box, as shown in Figure 11.41. This message box provides access to a dialog box that enables the user to choose from a list of applications that can operate on the type or convert the object's type.

## Activating Objects

Although selecting an object provides access to commands applicable to the object as a whole, it does not provide access to editing the content of the object. The object must be activated in order to provide user interaction with the internal content of the object. There are two basic models for activating objects: outside-in activation and inside-out activation.

## Outside-in Activation

*Outside-in activation* requires that the user choose an explicit activation command. Clicking, or some other selection operation, performed on an object that is already selected simply reselects that object and does not constitute an explicit action. The user activates the object by using a particular command such as Edit or Play, usually the object's default command. Shortcut actions that correspond to these commands, such as double-clicking or pressing a shortcut key, can also activate the object. Most OLE container applications employ this model because it allows the user to easily select objects and reduces the risk of inadvertently activating an object whose underlying code may take a significant amount of time to load and dismiss.

When supporting outside-in activation, display the standard pointer (northwest arrow) over an outside-in activated object within your container when the object is selected, but inactive. This indicates to the user that the outside-in object behaves as a single, opaque object. When the user activates the object, the object's application displays the appropriate pointer for its content. Use the registry to determine the object's activation command.

## Inside-out Activation

With *inside-out activation*, interaction with an object is direct; that is, the object is activated as the user moves the pointer over the extent of the object. From the user's perspective, inside-out objects are indistinguishable from native data because the content of the object is directly interactive and no additional action is necessary. Use this method for the design of objects that benefit from direct interaction, or when activating the object has little effect on performance or use of system resources.

Inside-out activation requires closer cooperation between the container and the object. For example, when the user begins a selection within an inside-out object, the container must clear its own selection so that the behavior is consistent with normal selection interaction. An object supporting inside-out activation controls the appearance of the pointer as it moves over its extent and responds immediately to input. Therefore, to select the object as a whole, the user selects the border, or some other handle, provided by the object or its container. For example, the container application can support selection techniques, such as region selection that select the object.

Although the default behavior for an OLE embedded object is outside-in activation, you can store information in the registry that indicates that an object's type (application class) is capable of inside-out activation (OLEMISC_INSIDEOUT) and prefers inside-out behavior (OLEMISC_ACTIVATEWHENVISIBLE). You can set these values in a **MiscStatus** subkey, under the **CLSID** subkey of the **HKEY_CLASSES_ROOT** key.

For more information about how to access OLEMISC_ IN-SIDEOUT and OLEMISC_ACTIVATE-WHENVISIBLE and the **IOleObject:: GetMiscStatus** function, see the OLE documentation included in the Win32 SDK.

## Container Control of Activation

The container application determines how to activate its component objects: either it allows the inside-out objects to handle events directly or it intercedes and only activates them upon an explicit action. This is true regardless of the capability or preference setting of the object. That is, even though an object may register inside-out activation, it can be treated by a particular container as outside-in. Use an activation style for your container that is most appropriate for its specific use and is in keeping with its own native style of activation so that objects can be easily assimilated.

Regardless of the activation capability of the object, a container should always activate its content objects of the same type consistently. Otherwise, the unpredictability of the interface is likely to impair its usability. Following are four potential container activation methods and when to use them.

| Activation method | When to use |
|---|---|
| Outside-in throughout | This is the most common design for containers that often embed large OLE objects and deal with them as whole units. Because many available OLE objects are not yet inside-out capable, most compound document editors support outside-in throughout to preserve uniformity. |
| Inside-out throughout | Ultimately, OLE containers will blend embedded objects with native data so seamlessly that the distinction dissolves. Inside-out throughout containers will become more feasible as increasing numbers of OLE objects support inside-out activation. |
| Outside-in plus inside-out preferred objects | Some containers may use an outside-in model for large, foreign embeddings but also include some inside-out preferred objects as though they were native objects (by supporting OLEMISC_ACTIVATEWHENVISIBLE). For example, an OLE document might present form control objects as inside-out native data while activating larger spreadsheet and chart objects as outside-in. |
| Switch between inside-out throughout and outside-in throughout | Visual programming and forms layout design applications may include design and run modes. In this type of environment, a container typically holds an object that is capable of inside-out activation (if not preferable) and alternates between outside-in throughout when designing and inside-out throughout when running. |

## OLE Visual Editing of OLE Embedded Objects

One of the most common uses for activating an object is editing its content in its current location. Supporting this type of in-place interaction is called *OLE visual editing*, because the user can edit the object within the visual context of its container.

Unless the container and the object both support inside-out activation, the user activates an embedded object for visual editing by selecting the object and choosing its Edit command, either from a drop-down or pop-up menu. You can also support shortcut techniques. For example, by making Edit the object's default operation, the user can double-click to activate the object for editing. Similarly, you can support pressing the ENTER key as a shortcut for activating the object.

Although earlier versions of OLE user interface documentation suggested the ALT+ENTER key combination to activate an object if the ENTER key was already assigned, this key combination is now the recommended shortcut key for the Properties command. Instead, support the pop-up menu shortcut key. This enables the user to activate the object by selecting the command from the pop-up menu.

When the user activates an OLE embedded object for visual editing, the user interface for its content becomes available and blended into its container application's interface. The object can display its frame *adornments*, such as row or column headers, handles, or scroll bars, outside the extent of the object and temporarily cover neighboring material. The object's application can also change the menu interface, which can range from adding menu items to existing drop-down menus to replacing entire drop-down menus. The object can also add toolbars, status bars, supplemental palette windows, and provide pop-up menus for selected content.

The degree of interface blending varies based on the nature of the OLE embedded object. Some OLE embedded objects may require extensive support and consequently result in dramatic changes to the container application's interface. Finer grain objects that emulate the native components of a container may have little or no need to make changes in the container's user interface. The container always determines the degree to which an OLE embedded object's interface can be blended with its own, regardless of the capability or preference of the OLE embedded object. A container application that provides its own interface for an OLE embedded object can suppress an OLE embedded object's own interface. Figure 11.16 shows how the interface might appear when its embedded worksheet is active.

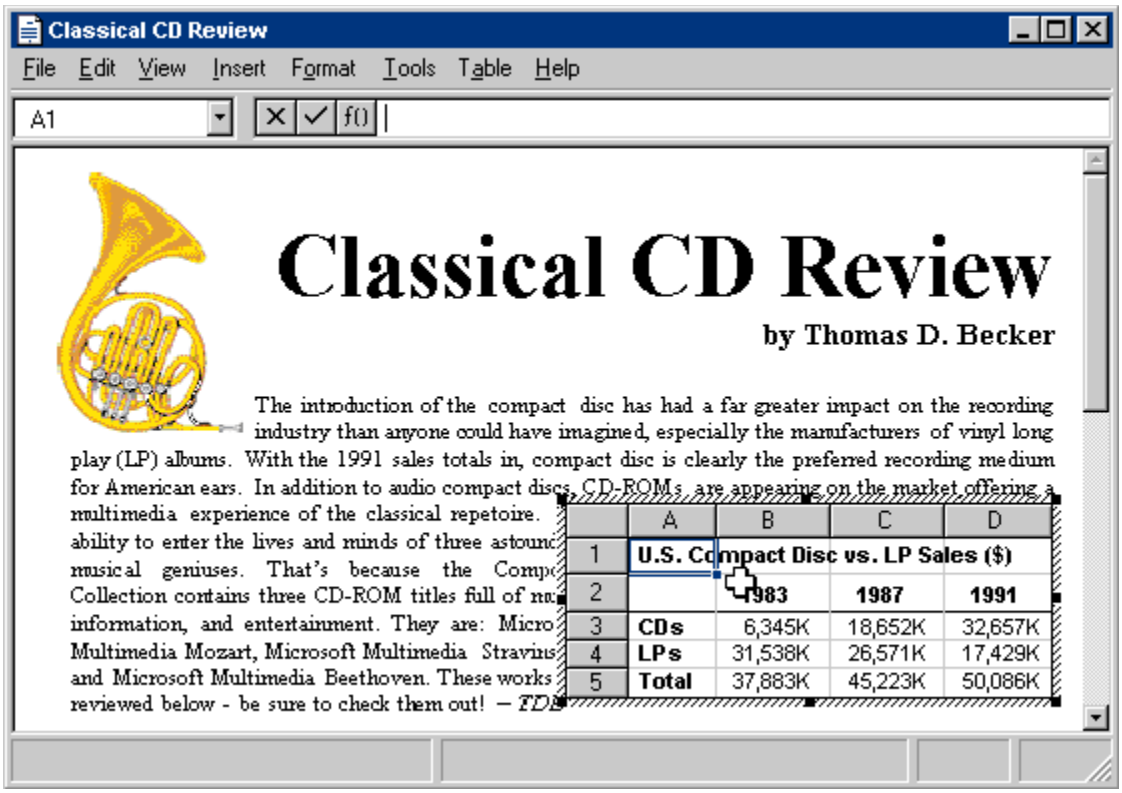



**Figure 11.16 An embedded worksheet activated for OLE visual editing**

When the user activates an OLE embedded object, avoid changing the view and position of the rest of the content in the window. Although it may seem reasonable to scroll the window and thereby preserve the content's position, doing so can disturb the user's focus, because the active object shifts down to accommodate a new toolbar and shifts back up when it is deactivated. An exception may be when the activation exposes an area in which the container has nothing to display. However, even in this situation, you may wish to render a visible region or filled area that corresponds to the background area outside the visible edge of the container so that activation keeps the presentation stable.

Activation does not affect the title bar. Always display the top-level container's name. For example, when the worksheet shown in Figure 11.16 is activated, the title bar continues to display the name of the document in which the worksheet is embedded and not the name of the worksheet. You can provide access to the name of the worksheet by supporting property sheets for your OLE embedded objects.

A container can contain multiply nested OLE embedded objects. However, only a single level is active at any one time. Figure 11.17 shows a document containing an active embedded worksheet with an embedded graph of its own. Clicking on the graph merely selects it as an object within the worksheet.

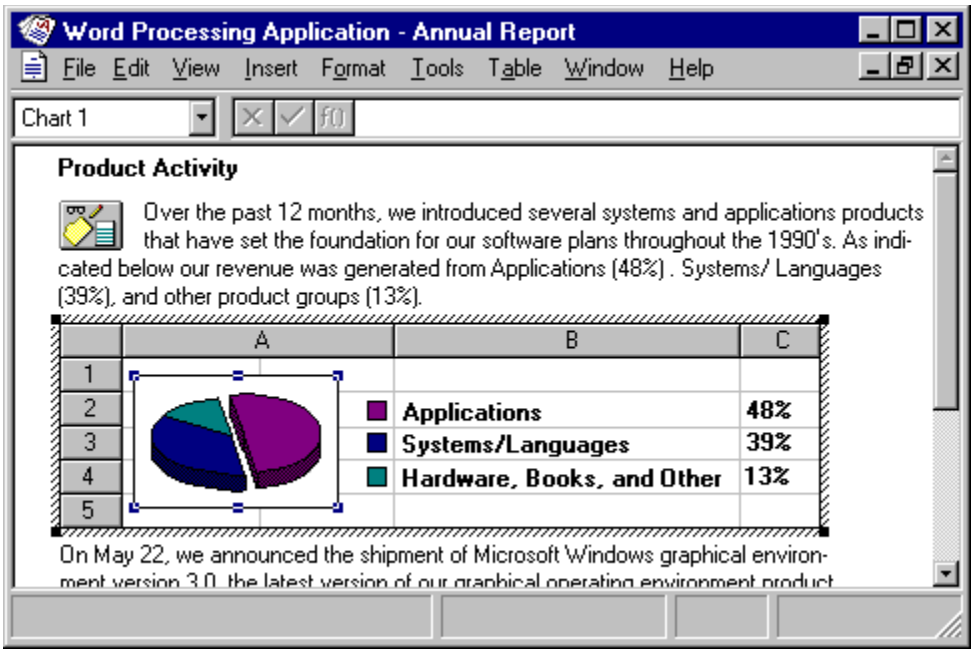



**Figure 11.17 A selected graph within an active worksheet**

Activating the embedded graph, for example, by choosing the graph's Edit command, activates the object for OLE visual editing, displaying the graph's menus in the document's menu bar. This is shown in Figure 11.18. At any given time, only the interface for the currently active object and the topmost container are presented; intervening parent objects do not remain visibly active.

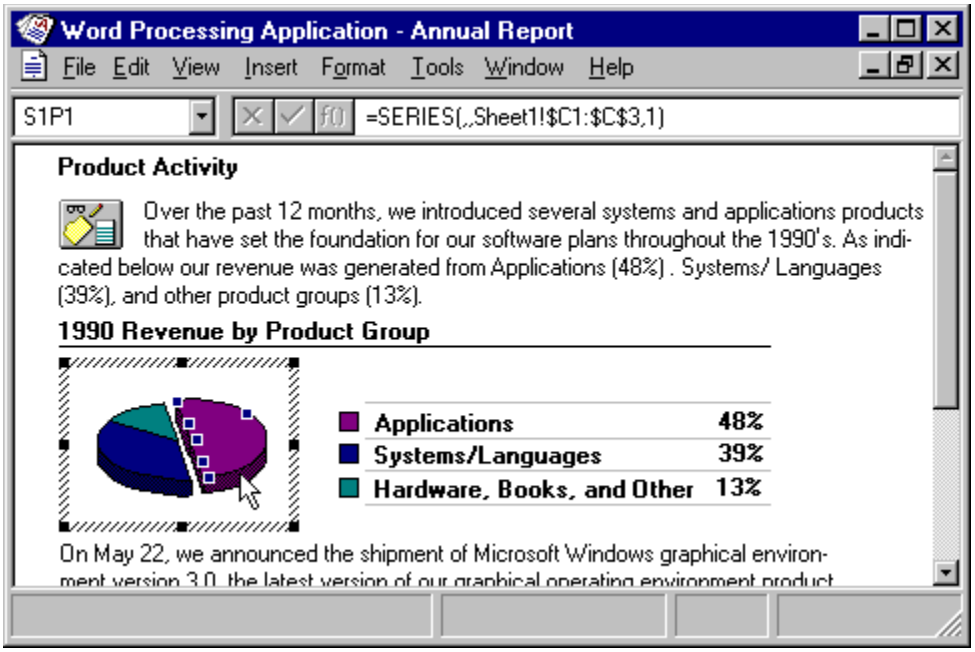



**Figure 11.18 An active graph within a worksheet**

An OLE embedded object should support OLE visual editing at any view magnification level because its container's view can be scaled arbitrarily. If an object cannot accommodate OLE visual editing in its container's current view scale, or if its container does not support OLE visual editing, open the object

into a separate window for editing. For more information about opening OLE embedded objects, see Opening OLE Embedded Objects.

For any user interaction outside the extent of an active object, such as when the user selects or activates another object in the container, deactivate the current object and give the focus to the new object. This is also true for an object that is nested in the currently active object. An OLE embedded object application should also support user deactivation when the user presses the ESC key, after which it becomes the selected object of its container. If the object uses the ESC key at all times for its internal operation, the SHIFT+ESC key should deactivate the object.

Edits made to an active object become part of the container immediately and automatically, just like edits made to native data. Consequently, do not display an “Update changes?” message box when the object is deactivated. Remember that the user can abandon changes to the entire container, embedded or otherwise, if the topmost container includes an explicit command that prompts the user to save or discard changes to the container’s file.

OLE embedded objects participate in the undo stack of the window in which they are activated. For more information about embedded objects and the undo stack, see Undo Operations for Active and Open Objects.

While Edit is the most common command for activating an OLE embedded object for OLE visual editing, other commands can also create such activation. For example, when the user carries out a Play command on a video clip, you can display a set of commands that allow the user to control the clip (Rewind, Stop, and Fast Forward). In this case, the Play command provides a form of OLE visual editing.

## The Active Hatched Border

If a container allows an OLE embedded object's user interface to change its user interface, then the active object's application displays a hatched border around itself to show the extent of the OLE visual editing context (shown in Figure 11.19). For example, if an active object places its menus in the topmost container's menu bar, display the active hatched border. The object's request to display its menus in the container's menu bar must be granted by the container application. If the object's menus do not appear in the menu bar (because the object did not require menus or the container refused its request for menu display), or the object is otherwise accommodated by the container's user interface, you need not display the hatched border. The hatched pattern is made up of 45-degree diagonal lines.

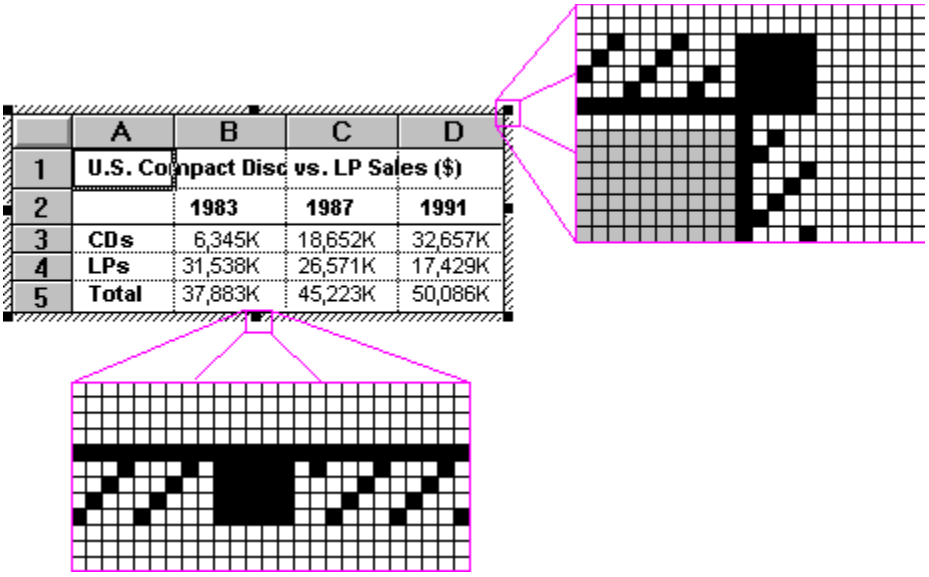



**Figure 11.19 Hatched border around active OLE embedded objects**

The active object takes on the appearance that is best suited for its own editing; for example, the object may display frame adornments, table gridlines, handles, and other editing aids. Because the hatched border is part of the object's territory, the active object defines the pointer that appears when the user moves the mouse over the border.

Clicking in the hatched pattern (and not on the handles) is interpreted by the object as clicking just inside the edge of the border of the active object. The hatched area is effectively a hot zone that prevents inadvertent deactivations and makes it easier to select the content of the embedded object.

## Menu Integration

As the user activates different objects, different commands need to be accessed in the window's user interface. The following classification of menus — primary container menu, workspace menu, and active object menus — separates the interface based on menu groupings. This classification enhances the usability of the interface by defining the interface changes as the user activates or deactivates different objects.

## Primary Container Menu

The topmost or primary container viewed in a primary window controls the work area of that window. If the primary container includes a menu bar, it supplies at least one menu that includes commands that apply to the primary container as an entire unit. For example, for document file objects, use a File menu for this purpose, as shown in Figure 11.20. This menu includes document and file level commands such as Open, Save, and Print. Always display the primary container menu in the menu bar at all times, regardless of which object is active.

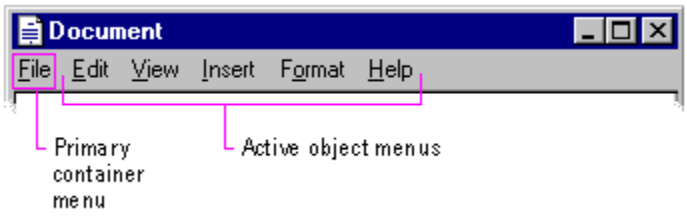



**Figure 11.20 OLE visual editing menu layout**

## Workspace Menu

An MDI-style application also includes a workspace menu (typically labeled “Window”) on its menu bar that provides commands for managing the document windows displayed within it, as shown in Figure 11.21. Like the primary container menu, the workspace menu should always be displayed, independent of object activation.

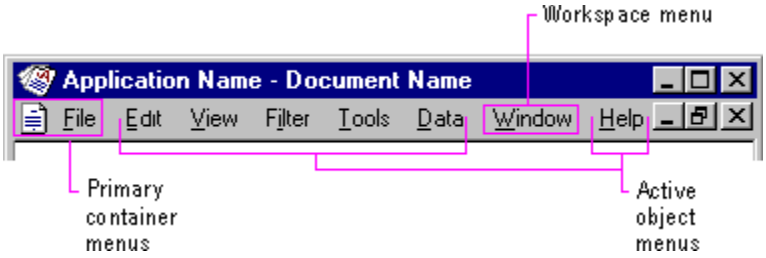



**Figure 11.21 OLE visual editing menu layout for MDI**

## Active Object Menus

Active objects can define menus that appear on the primary container's menu bar that operate on their content. Place commands for moving, deleting, searching and replacing, creating new items, applying tools, styles, and Help on these menus. As the name suggests, active object commands are executed by the currently active object and apply only within the extent of that object. If no embedded objects are active, but the window is active, the primary container should be considered the active object.

An active object's menus typically occupy the majority of the menu bar. Organize these menus following the same order and grouping that you display when the user opens the object into its own window. Avoid naming your active object menus File or Window, because primary containers often use those titles. Objects that use direct manipulation as their sole interface need not provide active object menus or alter the menu bar when activated.

The active object can display a View menu. However, when the object is active, include only commands that apply to the object. If the object's container requires its document or window-level "viewing" commands to be available while an object is active, place them on a menu that represents the primary container window's pop-up menu and on the Window menu — if present.

When designing the interface of selected objects within an active object, follow the same guidelines as that of a primary container and one of its selected OLE embedded objects; that is, the active object displays the commands of the selected object (as registered in the registry) either as submenus of its menus or as separate menus.

An active object also has the responsibility for defining and displaying pop-up menus for its content, with commands appropriate to apply to any selection within it. Figure 11.22 shows an example of a pop-up menu for a selection within an active bitmap image.

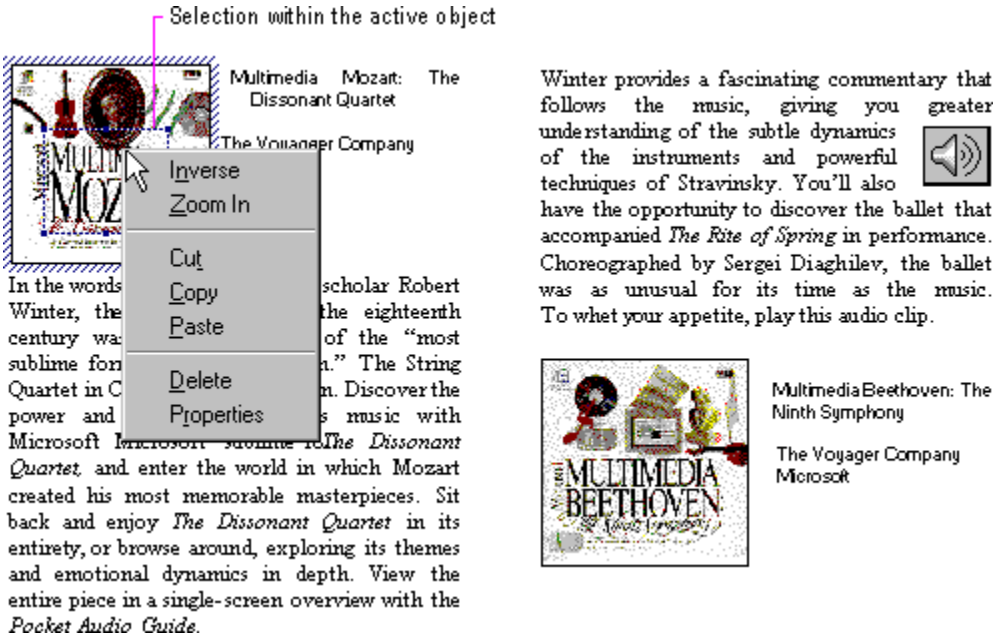



**Figure 11.22 Example of pop-up menu for a selection in an active object**

## Keyboard Interface Integration

In addition to integrating the menus, you must also integrate the access keys and shortcut keys used in these menus.

## Access Keys

The access keys assigned to the primary container's menu, an active object's menus, and MDI workspace menus should be unique. Following are guidelines for defining access keys for integrating these menu names:

- Use the first letter of the menu of the primary container as its access key character. Typically, this is "F" for File. Use "W" for a workspace's Window menu. Localized versions should use the appropriate equivalent.
- Use characters other than those assigned to the primary container and workspace menus for the menu titles of active OLE embedded objects. (If an OLE embedded object has previously existed as a standalone document, its corresponding application avoids these characters already.)
- Define unique access keys for an object's registered commands and avoid characters that are potential access keys for common container-supplied commands, such as Cut, Copy, Paste, Delete, and Properties.

Despite these guidelines, if the same access character is used more than once, pressing an ALT+*letter* combination cycles through each command, selecting the next match each time it is pressed. To carry out the command, the user must press the ENTER key when it is selected. This is standard system behavior for menus.

## Shortcut Keys

For primary containers and active objects, follow the shortcut key guidelines covered in this guide. In addition, avoid defining shortcut keys for active objects that are likely to be assigned to the container. For example, include the standard editing and transfer (Cut, Copy, and Paste) shortcut keys, but avoid File menu or system-assigned shortcut keys. There is no provision for registering shortcut keys for a selected object's commands.

For more information about defining shortcut keys, see Input Basics, and Keyboard Interface Summary.

If a container and an active object share a common shortcut key, the active object captures the event. That is, if the user activates an OLE embedded object, its application code directly processes the shortcut key. If the active object does not process the key event, it is available to the container, which has the option to process it or not. This applies to any level of nested OLE embedded objects. If there is duplication between shortcut keys, the user can always direct the key based on where the active focus is by activating that object. To direct a shortcut key to the container, the user deactivates an OLE embedded object — for example, by selecting in the container — but outside the OLE embedded object. Activation, not selection, of an OLE embedded object allows it to receive the keyboard events. The exception is inside-out activation, where activation results from selection.

## Toolbars, Frame Adornments, and Palette Windows

Integrating drop-down and pop-up menus is straightforward because they are confined within a particular area and follow standard conventions. Toolbars, frame adornments (as shown in Figure 11.23), and palette windows can be constructed less predictably, so it is best to follow a replacement strategy when integrating these elements for active objects. That is, toolbars, frame adornments, and palette windows are displayed and removed as entire sets rather than integrated at the individual control level — just like menu titles on the menu bar.

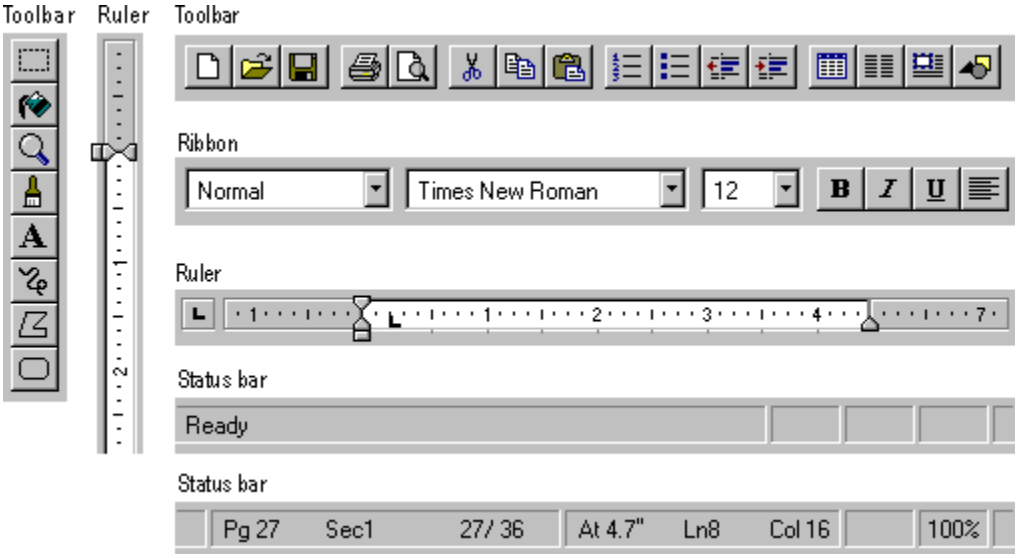



**Figure 11.23 Examples of toolbars, status bars, and frame adornments**

When the user activates an object, the object application requests a specific area from its container in which to post its tools. The container application determines whether to:

- Replace its tool (or tools) with the tools of the object, if the requested space is already occupied by a container tool.
- Add the object's tool (or tools), if a container tool does not occupy the requested space.
- Refuse to display the tool (or tools) at all. This is the least desirable method.

Toolbars, frame adornments, and palette windows are all basically the same interfaces — they differ primarily in their location and the degree of shared control between container and object. There are four locations in the interface where these types of controls reside, and you determine their location by their scope. Figure 11.24 shows possible positions for interface controls.

| Location | Description |
|---|---|
| Object frame | Place object-specific controls, such as a table header or a local coordinate ruler, directly adjacent to the object itself for tightly coupled interaction between the object and its interface. An object (such as a spreadsheet) can include scrollbars if its content extends beyond the boundaries of its frame. |
| Pane frame | Locate view-specific controls at the pane level. Rulers and viewing tools are common examples. |

| | |
|---|---|
| Document (primary container) window frame | Attach tools that apply to the entire document (or documents in the case of an MDI window) just inside any edge of its primary window frame. Popular examples include ribbons, drawing tools, and status lines. |
| Windowed | Display tools in a palette window — this allows the user to place them as desired. A palette window typically floats above the primary window and any other windows of which it is part. |


**Figure 11.24 Possible locations for interface controls**

When determining where to locate a tool area, avoid situations that cause the view to shift up and down as different-sized tool areas are displayed or removed as the user activates different objects. This can be disruptive to the user's task.

Because container tool areas can remain visible while an object is active, they are available to the user simply by interacting with them — this can reactivate the container application. The container determines whether to activate or leave the object active. If toolbar buttons of an active object represent a primary container or workspace commands, such as Save, Print, or Open, disable them.

For more information about the negotiation protocols used for activation, see the OLE documentation included in the Win32 SDK.

As the user resizes or scrolls its container's area, an active object and its toolbar or frame adornments placed on the object frame are clipped, as is all container content. These interface control areas lie in the same plane as the object. Even when the object is clipped, the user can still edit the visible part of the object in place and while the visible frame adornments are operational.

Some container applications scroll at certain increments that may prevent portions of an OLE embedded object from being visually edited. For example, consider a large picture embedded in a worksheet cell. The worksheet scrolls vertically in complete row increments; the top of the pane is always aligned with the top edge of a row. If the embedded picture is too large to fit within the pane at one time, its bottom portion is clipped and consequently never viewed or edited in place. In cases like this, the user can open the picture into its own window for editing.

Window panes clip frame adornments of nested embedded objects, but not by the extent of any parent object. Objects at the very edge of their container's extent or boundary potentially display adornments that extend beyond the bounds of the container's defined area. In this case, if the container displays items that extend beyond the edge, display all the adornments; otherwise, clip the adornments at the edge of the container. Do not temporarily move the object within its container just to accommodate the appearance of an active embedded object's adornments. A pane-level control can potentially be clipped by the primary (or parent, in the case of MDI) window frame, and a primary window adornment or control is clipped by other primary windows.

## Opening OLE Embedded Objects

The previous sections have focused on OLE visual editing — editing an OLE embedded object in place; that is, its current location is within its container. Alternatively, the user can also open embedded objects into their own window. This gives the user the opportunity of seeing more of the object or seeing the object in a different view state. Support this operation by registering an Open command for the object. When the user chooses the Open command of an object, it opens it into a separate window for editing, as shown in Figure 11.25.

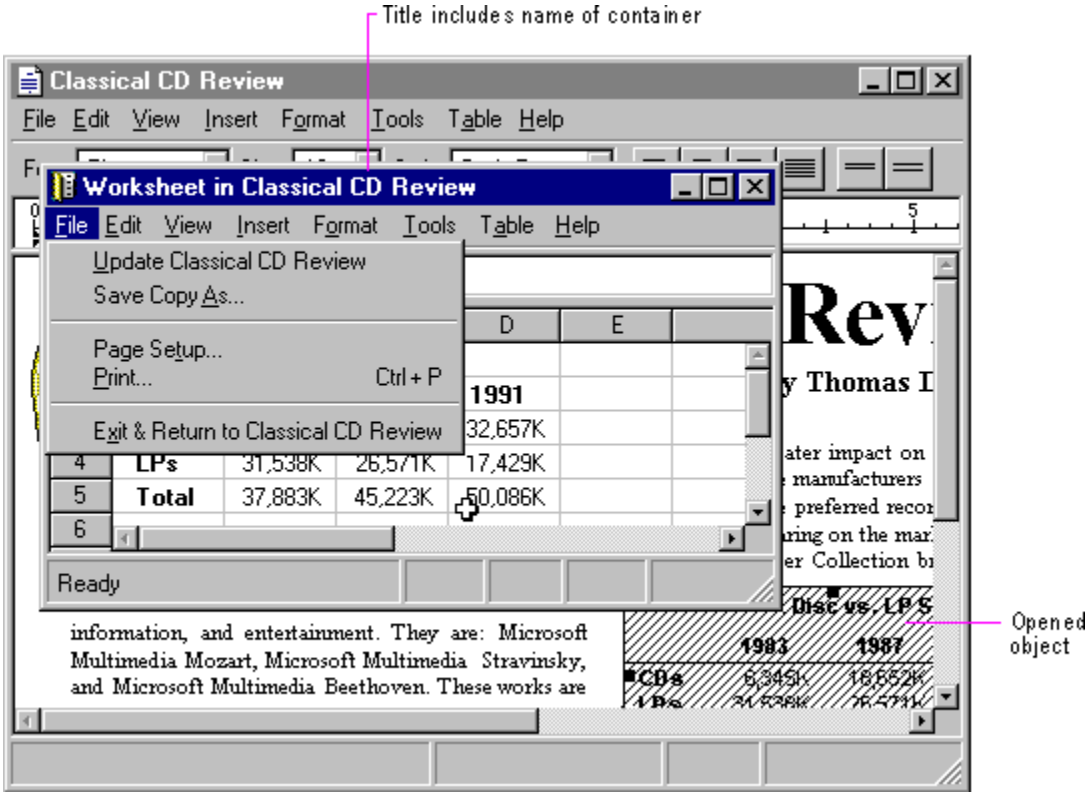



**Figure 11.25 An opened OLE embedded worksheet**

After opening an object, the container displays it masked with an “open” hatched (lines at a 45-degree angle) pattern that indicates the object is open in another window, as shown in Figure 11.26.

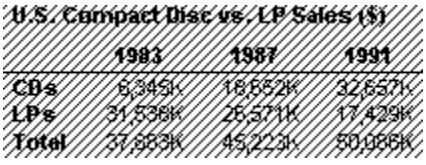



**Figure 11.26 An opened OLE embedded object**

Format the title text for the open object’s window as “*Object Name* in *Container Name*” (for example, “Sales Worksheet in Classical CD Review”). Including the container’s name emphasizes that the object in the container and the object in the open window are considered the same object.

This convention for the title bar text applies only when the user opens an OLE embedded object. When the user activates an OLE embedded object for visual editing, do not change the title bar text.

An open OLE embedded object represents an alternate window onto the same object within the container as opposed to a separate application that updates changes to the container document. Therefore, reflect edits immediately and automatically in the object in the window of its container. There is no need to display an update confirmation message upon exiting the open window. Nevertheless, you can still include an Update *Container Filename* command in the window of the open object to allow the user to request an update explicitly. This is useful if you cannot support frequent “real-time” image updates because of operational performance. In addition, when the user closes an open object’s window, automatically update its presentation in the container’s window. Provide a Close & Return To *Container Filename* or Exit & Return To *Container Filename* on the File menu replacing the Close or Exit command, as shown in Figure 11.25.

You can also include Import File and similar commands in the window of the open object. Treat importing a file into the window of the open embedded object the same as any change to the object.

If it has file operations, such as New or Open, remove these in the resulting window or replace them with commands, such as Import, to avoid severing the object’s connection with its container. The objective is to present a consistent conceptual model; the object in the opened window is always the same as the one in the container. You can replace the Save As command with a Save Copy As command that displays the Save As dialog box, but unlike Save As, Save Copy As does not make the copied file the active file.

When the user opens an object, it is the selected object in the container; however, the user can change the selection in the container afterwards. Like any selected OLE embedded object, the container supplies the appropriate selection appearance together with the open appearance, as shown in Figure 11.27.

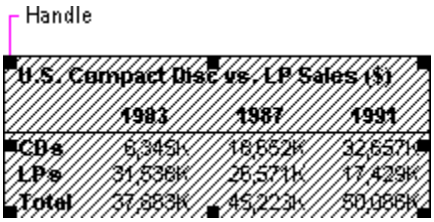



**Figure 11.27 A selected open OLE embedded object**

The selected and open appearances apply only to the object’s appearance on the display. If the user chooses to print the container while an OLE embedded object is open or active, use the presentation form of objects; neither the open nor active hatched pattern should appear in the printed document because neither pattern is part of the content.

While an OLE embedded object is open, it is still a functioning member of its container. It can still be selected or unselected, and can respond to appropriate container commands. At any time, the user may open any number of OLE embedded objects. When the user closes its container window, deactivate and close the windows for any open OLE embedded objects.

## Editing an OLE Linked Object

An OLE linked object can be stored in a particular location, moved or copied, and has its own properties. Container actions can be applied to the OLE linked object as a unit of content. So an OLE container supplies commands, such as Cut, Copy, Delete, and Properties, and interface elements such as handles, drop-down and pop-up menu items, and property sheets, for the OLE linked objects it contains.

The container also provides access to the commands that activate the OLE linked object, including the commands that provide access to content represented by the OLE linked object. These commands are the same as those that have been registered for the link source's type. Because an OLE linked object represents and provides access to another object that resides elsewhere, editing an OLE linked object always takes the user back to the link source. Therefore, the command used to edit an OLE linked object is the same as the command of its linked source object. For example, the menu of a linked object can include both Open and Edit if its link source is an OLE embedded object. The Open command opens the embedded object, just as carrying out the command on the OLE embedded object does. The Edit command opens the container window of the OLE embedded object and activates the object for OLE visual editing.

Figure 11.28 shows the result of opening a linked bitmap image of a horn. The image appears in its own window for editing. Note that changes made to the horn are reflected not only in its host container, the "Classical CD Review" document, but in every other document that contains an OLE linked object linked to that same portion of the "Horns" document. This illustrates both the power and the potential danger of using links in documents.

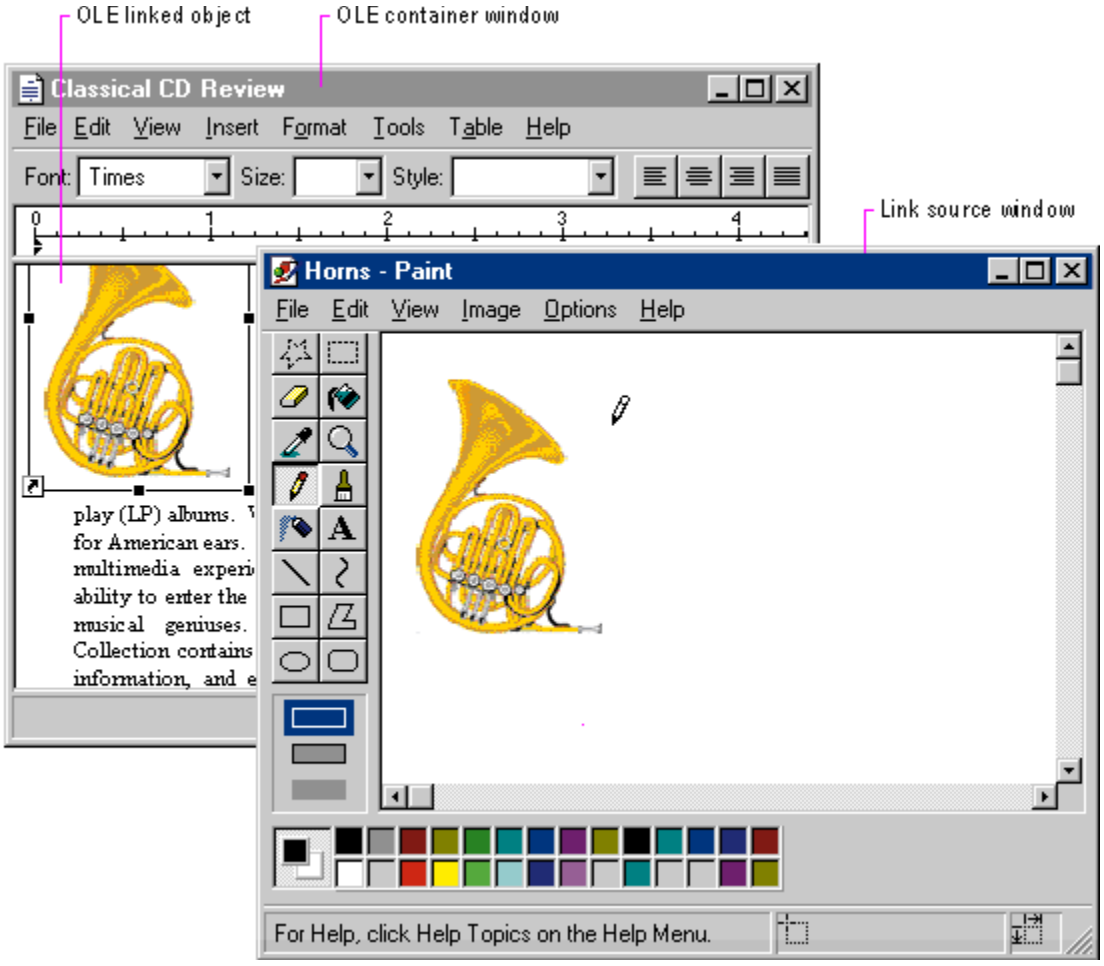



**Figure 11.28 Editing a link source**

At first glance, editing an OLE linked object seems to appear similar to an opened OLE embedded object; a separate primary window opens displaying the data. However, the container of an OLE linked object does not render the link representation using the open hatched pattern because the link source does not reside at this location. The OLE linked object is not the real object, only a stand-in that enables the source to be visually present in other locations. Editing the linked object is functionally identical to opening the link source. Similarly, the title bar text of the link source's window does not use the convention as an open OLE embedded object because the link source is an independent object. Therefore, the windows operate and close independently of each other. If the link source's window is already visible, the OLE linked object notifies the link source to activate, bringing the existing window to the top of the Z order.

Note that the container of the OLE linked object does display messages related to opening the link source. For example, the container displays a message if the link source cannot be accessed.

## Automatic and Manual Updating

When the user creates an OLE link, by default it is an automatic link; that is, whenever the source data changes, the link's visual representation changes without requiring any additional information from the user. Therefore, do not display an "Update Automatic Links Now?" message box. If the update takes a significant time to complete, you can display a message box indicating the progress of the update.

If users wish to exercise control over when links are updated, they can set the linked object's update property to manual. Doing so requires that the user choose an explicit command to update the link representation. The link can also be updated as a part of the link container's "update fields" or "recalc" action or other command that implies updating the presentation in the container's window.

## Operations and Links

The operations available for an OLE linked object are supplied by its container and its source. When the user chooses a command supplied by its container, the container application handles the operation. For example, the container processes commands such as Cut, Copy, or Properties. When the user chooses a command supplied (registered) by its source, the operation is conceptually passed back to the linked source object for processing. In this sense, activating an OLE linked object activates its source object.

In certain cases, the linked object exhibits the result of an operation; in other cases, the linked source object can be brought to the top of the Z order to handle the operation. For example, carrying out commands, such as Play or Rewind, on a link to a sound recording appear to operate on the linked object in place. However, if the user chooses a command to alter the link's representation of its source's content (such as Edit or Open), the link source is exposed and responds to the operation instead of the linked object itself.

A link can play a sound in place, but cannot support editing in place. For a link source to properly respond to editing operations, fully activate the source object (with all of its containing objects and its container). For example, when the user double-clicks a linked object whose default operation is Edit, the source (or its container) opens, displaying the linked source object ready for editing. If the source is already open, the window displaying the source becomes active. This follows the standard convention for activating a window already open; that is, the window comes to the top of the Z order. You can adjust the view in the window, scrolling or changing focus within the window, as necessary, to present the source object for easy user interaction. The linked source window and linked object window operate and close independently of each other.

If a link source is contained within a read-only document, display a message box advising the user that edits cannot be saved to the source file.

## Types and Links

An OLE linked object includes a cached copy of its source's type at the time of the last update. When the type of a linked source object changes, all links derived from that source object contain the old type and operations until either an update occurs or the linked source is activated. Because out-of-date links can potentially display obsolete operations to the user, a mismatch can occur. When the user chooses a command for an OLE linked object, the linked object compares the cached type with the current type of the linked source. If they are the same, the OLE linked object forwards the operation on to the source. If they are different, the linked object informs its container. In response, the container can either:

- Carry out the new type's operation, if the operation issued from the old link is syntactically identical to one of the operations registered for the source's new type.
- Display a message box, if the issued operation is no longer supported by the link source's new type (as shown in Figure 11.44).

In either case, the OLE linked object adopts the source's new type, and subsequently the container displays the new type's operations in the OLE linked object's menu.

## Link Management

An OLE linked object includes properties such as: the name of its source, its source's type, and the link's updating basis, which is either automatic or manual. An OLE linked object also has a set of commands related to these properties. It is the responsibility of the container of the linked object to provide the user access to these commands and properties. To support this, an OLE container provides a property sheet for all of its OLE objects. You can optionally also include a Links dialog box for viewing and altering the properties of several links simultaneously.

## Accessing Properties of OLE Objects

Like other types of objects, OLE embedded and linked objects have properties. The container of an OLE object is responsible for providing the user interface for access to the object's properties. The following sections describe how to provide user access to the properties of OLE objects.

## The Properties Command

Design OLE containers to include a Properties command and property sheets for any OLE objects it contains. If the container application already includes a Properties command for its own native data, you can also use it to support selected OLE embedded or linked objects. Otherwise, add the command to the drop-down and pop-up menu you provide for accessing the other commands for the object, preceded by a menu separator, as shown in Figure 11.29.

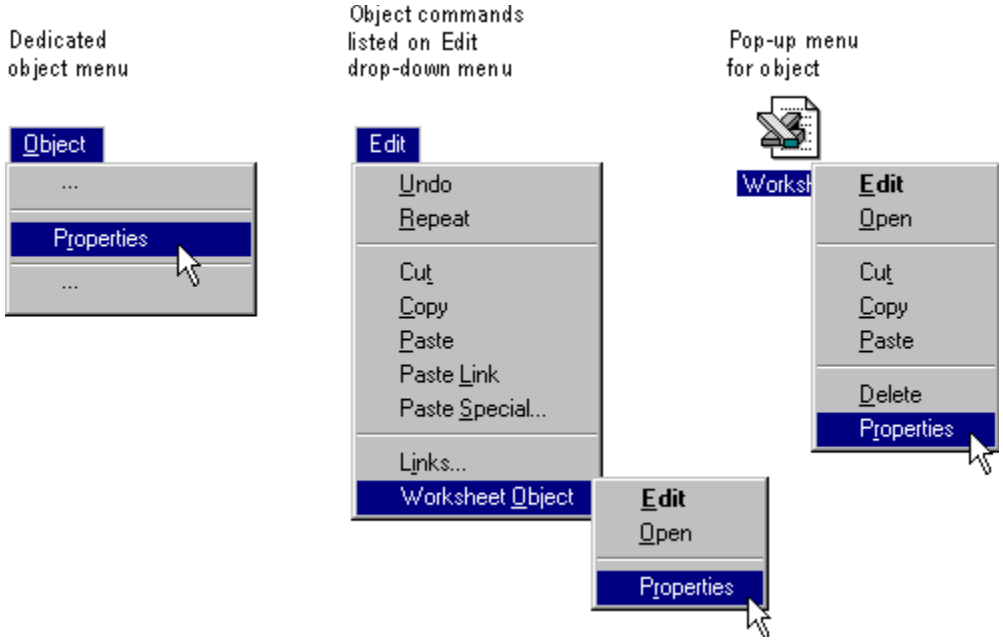
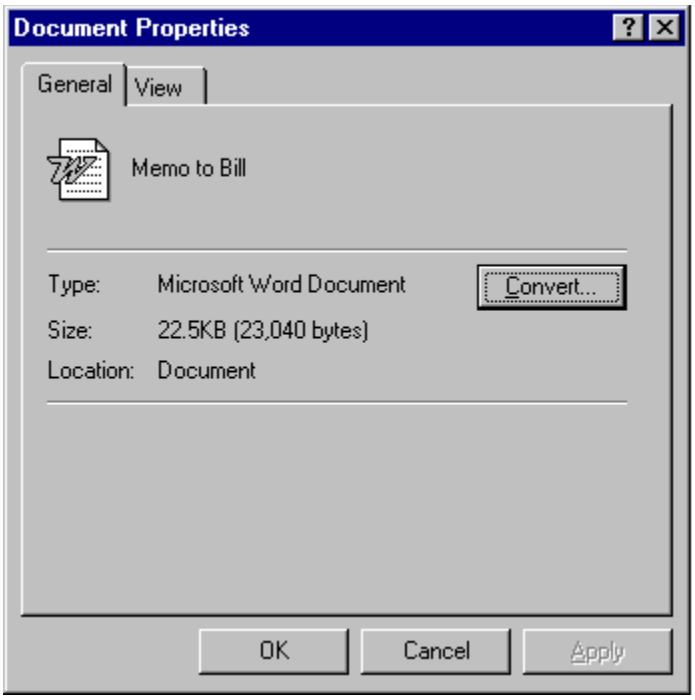



**Figure 11.29 The Properties command**

When the user chooses the Properties command, the container displays a property sheet containing all the salient properties and values, organized by category, for the selected object. Figure 11.30 shows examples property sheet pages for an OLE object.



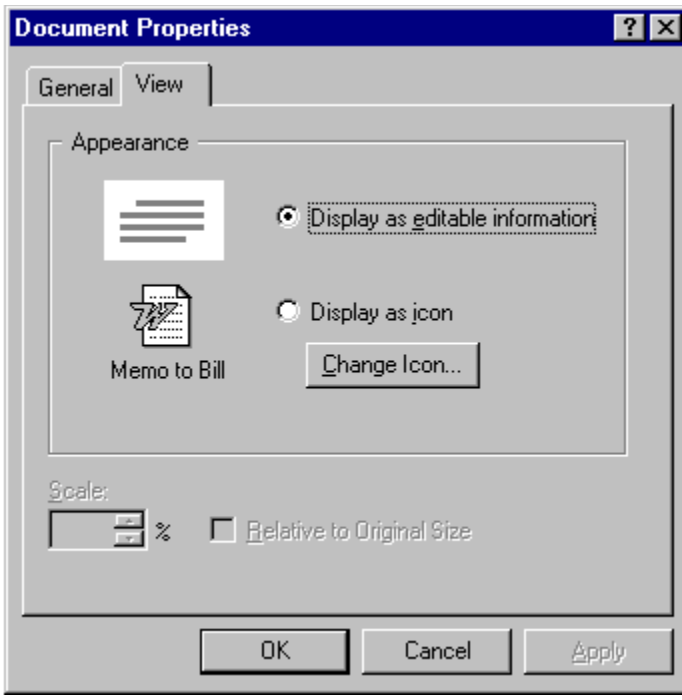




**Figure 11.30 OLE embedded object property sheet**

Follow the format the system uses for property sheets and the conventions outlined in this guide. Use the short type name in the title bar; for an OLE linked object, precede the name with the word “Linked,” as in “Linked Worksheet.” Include a General property page displaying the icon, name, type, size, and location of the object. Also include a Convert command button to provide access to the type conversion dialog box. On a View page, display properties associated with the view and presentation of the OLE object within the container. These include scaling or position properties and whether to display the object in its content presentation or as an icon. The Display As Icon field includes a Change Icon command button that allows the user to customize the icon presentation of the object. The Change Icon dialog box is shown in Figure 11.31.

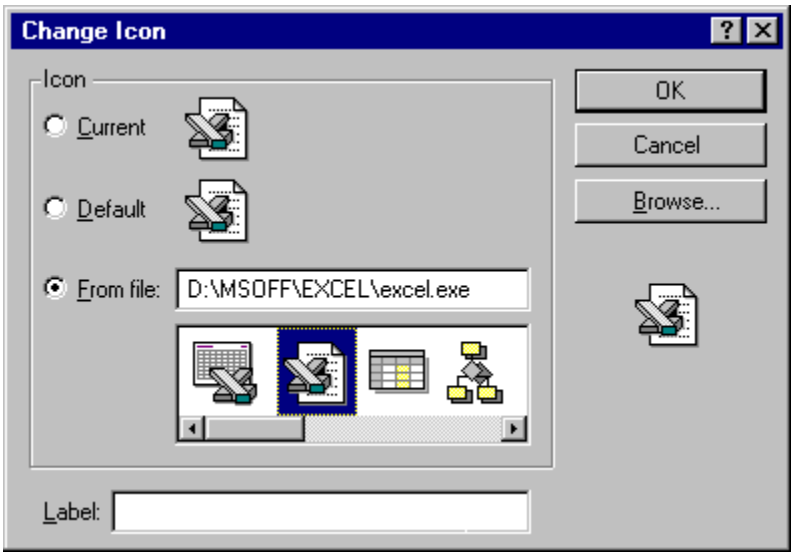



**Figure 11.31 The Change Icon dialog box**

For OLE linked objects, also include a Link page in its property sheet containing the essential link parameters and commands, as shown in Figure 11.32.

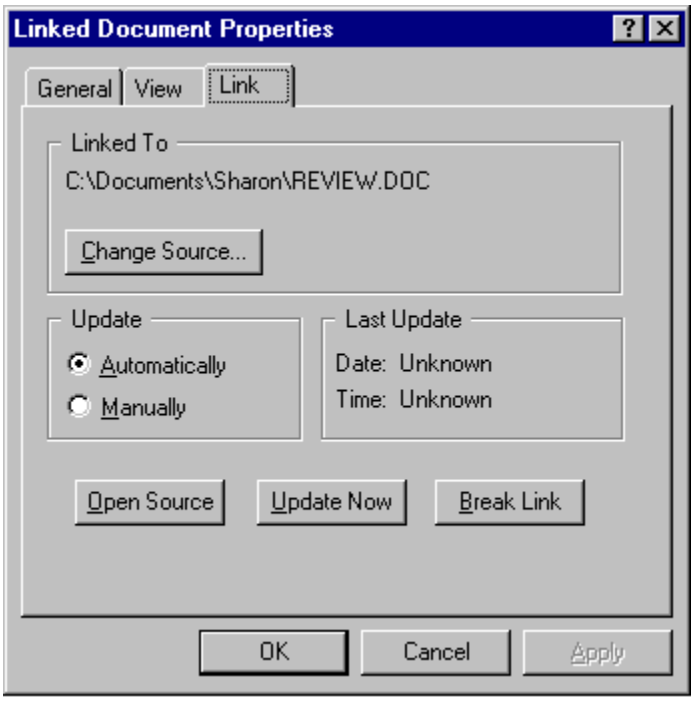



**Figure 11.32 The Link page for the property sheet of an OLE linked object**

For the typical OLE link, include the source name, the Update setting (automatic or manual), the Last Update timestamp, and command buttons that provide the following link operations:

- Break Link effectively disconnects the selected link.
- Update Now forces the selected link to connect to its sources and retrieve the latest information.
- Open Source opens the link source for the selected link.
- Change Source invokes a dialog box similar to the common Open dialog box to allow the user to respecify the link source.

## The Links Command

In addition to property sheets, OLE containers can optionally include a Links command that provides access to a dialog box for displaying and managing multiple links. Figure 11.33 shows the Links dialog box. The list box in the dialog box displays the links in the container. Each line in the list contains the link source's name, the link source's object type (short type name), and whether the link updates automatically or manually. If a link source cannot be found, "Unavailable" appears in the update status column.

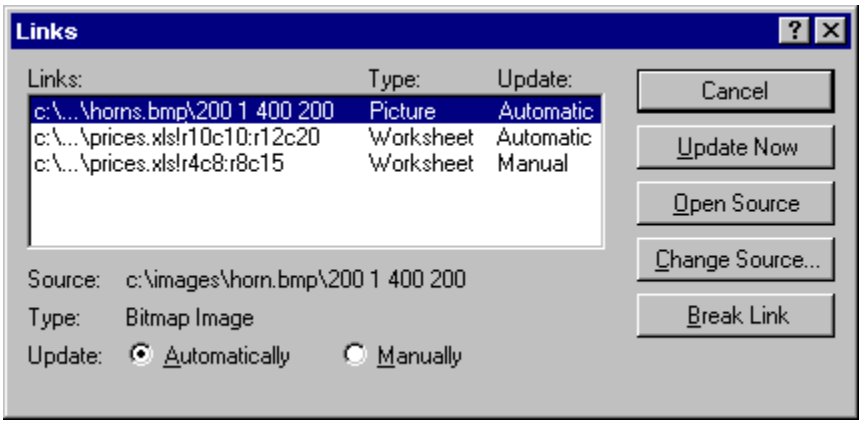



**Figure 11.33 The Links dialog box**

If the user chooses the Links command when the current selection includes a linked object (or objects), display that link (or links) as selected in the Links dialog box and scroll the list to display the first selected link at the top of the list box.

Allow 15 characters for the short type name field, and enough space for Automatic and Manual to appear completely. As the user selects each link in the list, its type, name, and updating basis appear in their entirety at the bottom of the dialog box. The dialog box also includes link management command buttons included in the Link page of OLE linked object property sheets: Break Link, Update Now, Open Source, and Change Source.

Define the Open Source button to be the default command button when the input focus is within the list of links. Support double-clicking an item in the list as a shortcut for opening that link source.

Clicking the Change Source button displays a version of the Open dialog box that allows the user to change the source of a link by selecting a file or typing a filename. If the user enters a source name that does not exist and chooses the default button, a message box is displayed with the following message, as shown in Figure 11.34.

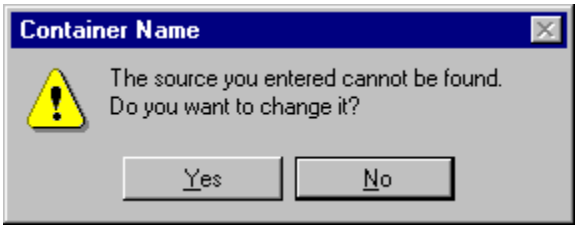



**Figure 11.34 A message box for an invalid source**

If the user chooses Yes, display the Change Source dialog box to correct the string. If the user chooses No, store the unparsed display name of the link source until the user links successfully to a newly created object that satisfies the dangling reference. The container application can also choose to allow the user to connect only to valid links.

If the user changes a link source or its directory, and other linked objects in the same container are connected to the same original link source, the container may offer the user the option to make the changes for the other references. To support this option, use the message box, as shown in Figure 11.35.

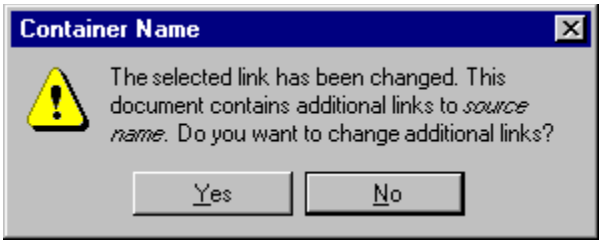



**Figure 11.35 Changing additional links with the same source**

## Converting Types

Users may want to convert an object's type, so they can edit the object with a different application. To support the user's converting an OLE object from its current type to another registered type, provide a Convert dialog box, as shown in Figure 11.36. The user accesses the Convert dialog box by including a Convert button beside the Type field in an object's property sheet.

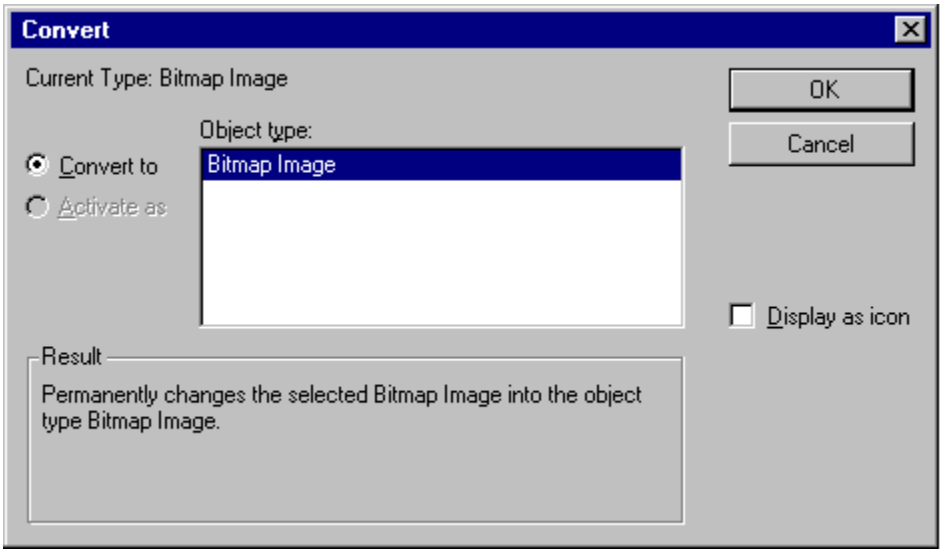



**Figure 11.36 The Convert dialog box**

This dialog box displays the current type of the object and a list box with all possible conversions. This list is composed of all types registered as capable of reading the selected object's format, but this does not necessarily guarantee the possibility of reverse conversion. If the user selects a new type from the list and chooses the OK button, the selected object is converted immediately to the new type. If the object is open, the container closes it before beginning the conversion.

Previous guidelines recommended including a Convert command on the menu for a selected OLE object. You may continue to support this; however, providing access through a button in the property sheet of the object is the preferred method.

Make sure the application that supplies the conversion does so with minimal impact in the user interface. That is, avoid displaying the application's primary window, but do provide a progress indicator message box with appropriate controls so that the user can monitor or interrupt the conversion process.

If the conversion of the type could result in any lost data or information, the application you use to support the type conversion should display a warning message box indicating that data will be lost and request confirmation by the user before continuing. Make the message as specific as possible about the nature of the information that might be lost; for example, "Text properties will not be preserved." If the conversion will result in no data loss, the warning message is not necessary.

In addition to converting a type, the Convert dialog box offers the user the option to change the type association for the object by choosing the Activate As option. When the user chooses this option, selects a type from the list, and chooses the OK button, the object's type is now treated as the new type. This differs from type conversion in that the object's type remains the same, but its activation command is now handled by a different application. It also differs in that converting a type only affects the object that is converted. Changing the activation association of a single object of the type, changes it for all OLE embedded objects of that type. For example, converting a rich-text format document to a text document only affects the converted document. However, if the user chooses the Activate As option to change the association for the rich-text format object so they will be activated as a text object

(that is, by the same application registered for editing text objects), all OLE embedded rich-text format objects will be also be activated this way.

At the bottom of the Convert dialog box, text describes the outcome of the choices the user selects. Table 11.3 outlines the syntax of the descriptive text to use within the Convert dialog box.

**Table 11.3 Descriptive Text for Convert Dialog Box**

| Function | Resulting text |
|---|---|
| Convert the selected object's type to a new type. | "Permanently changes the selected *Existing Type Name* object to a *New Type Name* object." |
| Convert the selected object's type to a new type and display the object as an icon. | "Permanently changes the selected *Existing Type Name* object to a *New Type Name* object. The object will be displayed as an icon." |
| No type change (the selected type is the same as its existing type). | "The *New Type Name* you selected is the same as the type of the selected object, so its type will not be converted." |
| Change the activation association for the selected object's type. | "Every *Existing Type Name* object will be activated as a *New Type Name* object, but not be converted to the new type." |
| Change the activation association for the selected object's type and display the object as an icon. | "Every *Existing Type Name* object will be activated as a *New Type Name* object, but converted to the new type. The selected object will be displayed as an icon. |

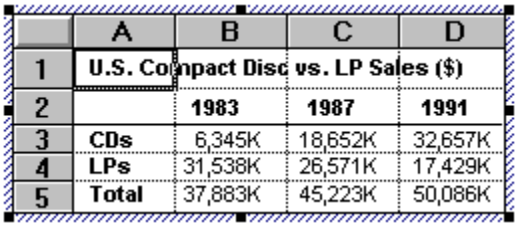
Disable the Convert option for a linked object because conversion for a link must occur on the link source. Also disable Activate As option if no types are registered for alternative activation. If the user can neither convert nor change the activation association, disable the Convert command that displays this dialog box.

## Using Handles

A container displays handles for an OLE embedded or linked object when the object is selected individually. When an object is selected and not active, only the scaling of the object (its cached metafile) can be supported. If a container uses handles for indicating selection but does not support scaling of the image, use the hollow form of handles.

For more information about the appearance of handles, see Visual Design.

When an OLE embedded object is activated for OLE visual editing, it displays its own handles. Display the handles within the active hatched pattern, as shown in Figure 11.37.



| | A | B | C | D |
|---|---|---|---|---|
| 1 | U.S. Co | mpact Disc vs. LP Sales ($) | | |
| 2 | | 1983 | 1987 | 1991 |
| 3 | CDs | 6,345K | 18,652K | 32,657K |
| 4 | LPs | 31,538K | 26,571K | 17,429K |
| 5 | Total | 37,883K | 45,223K | 50,086K |

**Figure 11.37 An active OLE embedded object with handles**

The interpretation of dragging the handle is defined by the OLE embedded object's application. The recommended operation is cropping, where you expose more or less of the OLE embedded object's content and adjust the viewport. If cropping is inappropriate or unsupportable, use an operation that better fits the context of the object or simply support scaling of the object. If no operation is meaningful, but handles are required to indicate selection while activated, use the hollow handle appearance.

## Undo Operations for Active and Open Objects

Because different objects (that is, different underlying applications) take control of a window during OLE visual editing, managing commands like Undo or Redo present a question: how are the actions performed within an edited OLE embedded object reconciled with actions performed on the native data of the container with the Undo command? The recommended undo model is a single undo stack per open window — that is, all actions that can be reversed, whether generated by OLE embedded objects or their container, accumulate on the same undo state sequence. Therefore, choosing Undo from either the container's menus or an active object's menus reverses the last undoable action performed in that open window, regardless of whether it occurred inside or outside the OLE embedded object. If the container has the focus and the last action in the window occurred within an OLE embedded object, when the user chooses Undo, activate the embedded object, reverse the action, and leave the embedded object active.

The same rule applies to open objects — that is, objects that have been opened into their own window. Because each open window manages a single stack of undoable states, actions performed in an open object are local to that object's window and consequently must be undone from there; actions performed in the open object (even if they create updates in the container) do not contribute to the undo state of the container.

Carrying out a registered command of a selected, but inactive, object (or using a shortcut equivalent) is not a reversible action; therefore, it does not add to a container's undo stack. For example, if the user opens an object, this action cannot be undone from its container. The resulting window must be closed directly to remove it.

Figure 11.38 shows two windows: container Window A, which has an active OLE embedded object, and an open embedded object in Window B. Between the two windows, nine actions have been performed in the order and at the location indicated by the numbers. The resulting undo stacks are displayed beneath the windows.

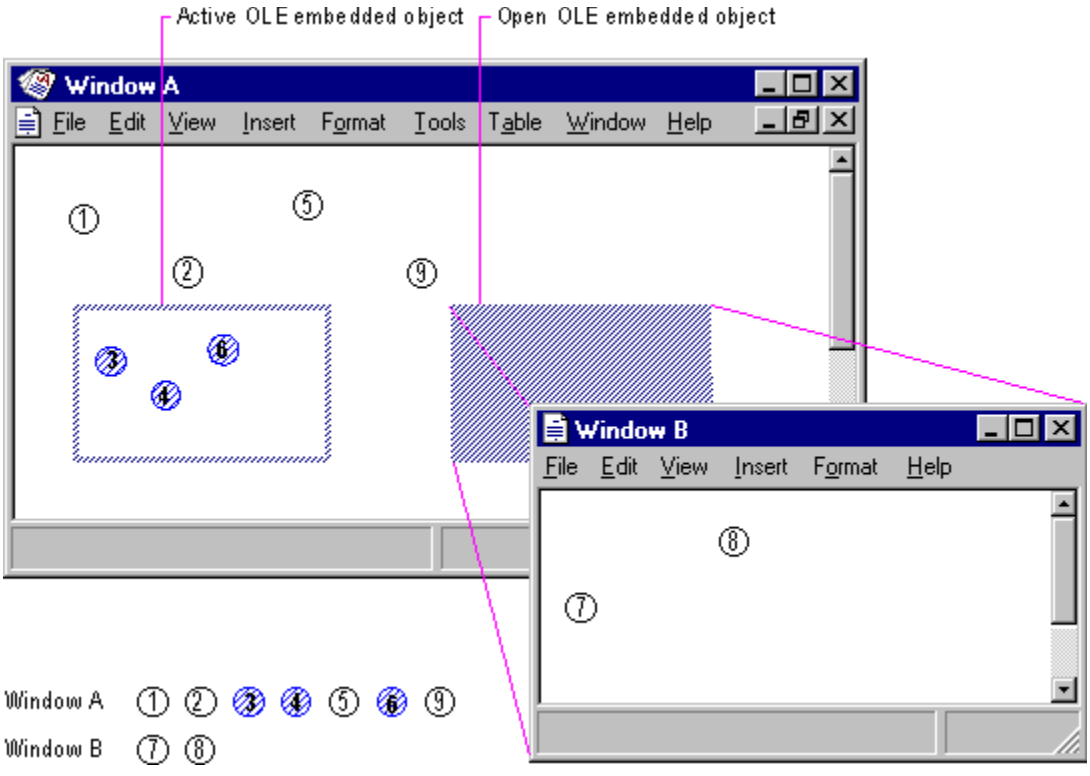



**Figure 11.38 Undo stacks for active and open OLE embedded objects**

The sequence of undo states shown in Figure 11.38 does not necessarily imply an *n*-level undo. It is

merely a timeline of actions that can be undone at 0, 1, or more levels, depending on what the container-object cooperation supports.

The active object actions and native data actions within Window A have been serialized into the same stack, while the actions in Window B have accumulated onto its own separate stack.

The actions discussed so far apply to a single window, not to actions that span multiple windows, such as OLE drag and drop. For a single action that spans multiple windows, the ideal design allows the user to undo the action from the last window involved. This is because, in most cases, the user focuses on that window when desiring to reverse the action. So if the user drags and drops an item from Window A into Window B, the action appends to Window B's undo thread, and undoing it undoes the entire OLE drag and drop operation. Unfortunately, the system does not support multiple window undo coordination. So for a multiple window action, create independent undo actions in each window involved in the action.

## Displaying Messages

This section includes recommendations about other messages to display for OLE interaction using message boxes and status line messages. Use the following messages in addition to those described in earlier topics.

The system supplies most of the message boxes described in this topic. For more information about how to support these, see the OLE documentation included in the Win32 SDK.

## Object Application Messages

Display the following messages to notify the user about situations where an OLE object's application is not accessible.

## Object's Application Cannot Run Standalone

Some OLE objects are designed to be used only as components within a container and have no value in being opened directly. If the user attempts to open or run an OLE object's application that cannot run as a standalone application, display the message box shown in Figure 11.39.


**Figure 11.39 Object's application cannot be run standalone message**

## Object's Application Busy

An object's application can be running, but busy for several reasons. For example, it can be busy printing, waiting for user input to a modal message box, or the application has stopped responding to the system. If the object's application is busy, display the message box shown in Figure 11.40.

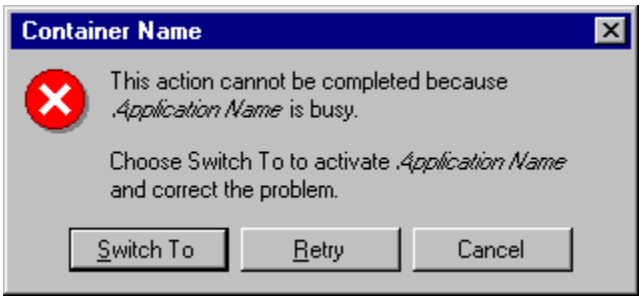



**Figure 11.40 Object's application is busy message**

## Object's Application Unavailable

If the user attempts to activate an object and the container cannot locate the requested object's application, for example, because the object's type is not registered or because the network server where the application resides is unavailable, display the message box shown in Figure 11.41.

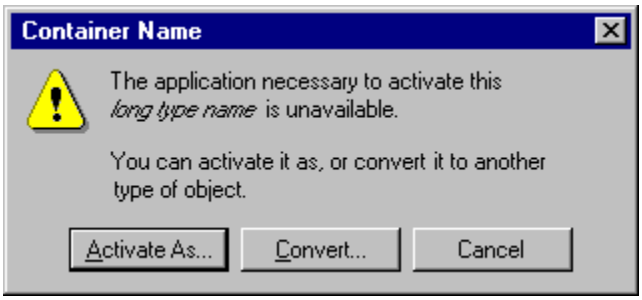



**Figure 11.41 Object's application is unavailable message**

Choosing the Activate As button displays the Convert dialog box preset with the Activate As option and a list of current types the user can use to associate with activating the object. Choosing the Convert button displays the Convert dialog box with the Convert option set, and the list of types the user can choose to change the type of the object. Ideally, an application that registers the type should be able to read and write that format without any loss of information. If it cannot preserve the information of the original type, the application handling the type emulation displays a message box warning the user about what information it cannot preserve and optionally allows the user to convert the object's type.

If a container supports inside-out activation for an object, display this message when the user tries to interact with that object, not when its container is opened. This avoids the display of the message to the user who only intends to view the content.

## OLE Linked Object Messages

Display the following messages to notify the user about situations related to interaction with OLE linked objects.

## Link Source Files Unavailable

When a container requests an update for its OLE linked objects, either because the user chooses an explicit Update command or as the result of another action such as a Recalc operation that forces an update, if the link source files for some OLE links are unavailable to provide the update, display the message box shown in Figure 11.42.

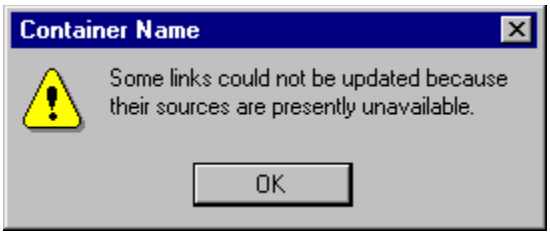



**Figure 11.42 Link source files are unavailable message**

When the user chooses the OK button, close the dialog box without updating the links.

Optionally, if you want to support the user changing the source, you supply your own message box that also includes a Properties button, a Links button, or both in the message box. Choosing the Properties button displays the property sheet for the link (see Figure 11.32) with “Unavailable” in the Update field. The user can then use the Change Source button to search for the file or choose other commands related to the link. When the user chooses this Links button, display your Links dialog box, following the same conventions as for the property sheet.

Similarly, if the user issues a command to an OLE linked object with an unavailable source, display the warning message shown in Figure 11.43.

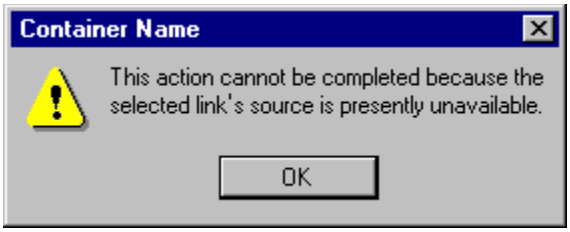



**Figure 11.43 Selected link source is unavailable message**

You can also supply your own message if you want to provide a Properties or Links button that enables the user to change the source. Display the OLE linked object’s update status as “Unavailable.”

## Link Source Type Changed

If a link source's type has changed, but it is not yet reflected for an OLE linked object, and the user chooses a command that does not support the new type, display the message box shown in Figure 11.44.

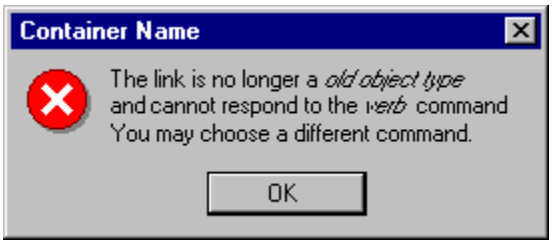



**Figure 11.44 Link source's type has changed message**

## Link Updating

While links are updating, display the progress indicator message box shown in Figure 11.45. The Stop button interrupts the update process and prevents any updating of additional links.

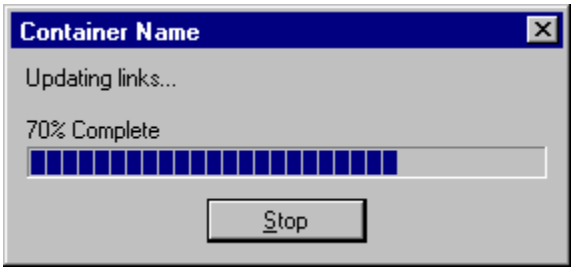



**Figure 11.45 Progress indicator while links update message**

## Status Line Messages

Table 11.4 lists suggested status line messages for commands on the primary container menu (commonly the File menu) of an opened object.

**Table 11.4 Primary Container Menu Status Line Messages**

| Command | Status line message |
|---|---|
| Update *Container-Document* | Updates the appearance of this *Type Name* in *Container-Document*. |
| Close & Return to *Container-Document* | Closes Object Name and returns to *Container-Document*. |
| Save Copy As | Saves a copy of *Type Name* in a separate file. |
| Exit & Return to *Container-Document* | Exits *Object Application* and returns to *Container-Document*. |

If the open object is within an MDI application with other open documents, the Exit & Return To command should simply be “Exit”. There is no guarantee of a successful Return To *Container-Document* after exiting, because the container might be one of the other documents in that MDI instance.

Table 11.5 lists the recommended status line messages for the Edit menu of containers of OLE embedded and linked objects.

**Table 11.5 Edit Menu Status Line Messages**

| Command | Status line message |
|---|---|
| Paste *Object Name*[1] | Inserts the content of the Clipboard as *Object Name*. |
| Paste Special | Inserts the content of the Clipboard with format options. |
| Paste Link [to *Object Name*[1]] | Inserts a link to *Object Name*. |
| Paste Shortcut [to *Object Name*[1]] | Inserts a shortcut icon to *Object Name*. |
| Insert Object | Inserts a new object. |
| [Linked] *Object Name*[1] [Object] | Applies the following commands to *Object Name*. |
| [Linked] *Object Name Command*[1] [Object] | Varies based on command. |
| [Linked] *Object Name* Properties[1] [Object] | Allows properties of *Object Name* to be viewed or modified. |
| Links | Allows links to be viewed, updated, opened, or removed. |

[1] *Object Name* may be either the object’s short type name or its filename.

Table 11.6 lists other related status messages.

**Table 11.6 Other Status Line Messages**

| Command | Status line message |
|---|---|
| Show Objects | Displays the borders around objects (toggle). |
| *Select Object* (when the user selects an object) | Double-click or press ENTER to *Default Command Object Name*. |

The default command stored in the registry contains an ampersand character (&) and the access key indicator; these must be stripped out before the verb is displayed on the status line.

# Introduction

Online user assistance is an important part of a product's design and can be supported in a variety of ways, from automatic display of information based on context to commands that require explicit user selection. Its content can be composed of contextual, procedural, explanatory, reference, or tutorial information. But user assistance should always be simple, efficient, and relevant so that a user can obtain it without becoming lost in the interface. This topic provides a description of the system support to create common online forms of user assistance support and guidelines for implementation. For more information about authoring Help files, see the documentation included in the Microsoft Win32 Software Development Kit (SDK).

# Contextual User Assistance

A contextual form of user assistance provides information about a particular object and its context. It answers questions such as “What is this?” and “Why would I use it?” This section covers some of the basic ways to support contextual user assistance in your application.

## Context-Sensitive Help

The What's This? command supports a user obtaining contextual information about any object on the screen, including controls in property sheets and dialog boxes. This form of contextual user assistance is referred to as *context-sensitive Help*. As shown in Figure 12.1, you can support user access to this command by including:

- A What's This? command from the Help drop-down menu of a primary window.
- A What's This? button on a toolbar.
- A What's This? button on the title bar of a secondary window.
- A What's This? command included on the pop-up menu for the specific object.

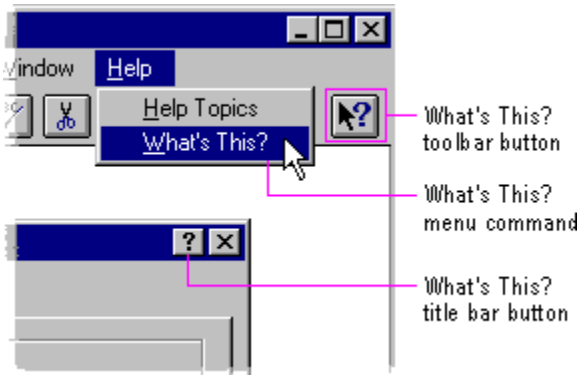



**Figure 12.1 Different methods of accessing What's This?**

Design your application so that when the user chooses the What's This? command from the Help drop-down menu or clicks a What's This? button, the system is set to a temporary mode. Change the pointer's shape to reflect this mode change, as shown in Figure 12.2. The SHIFT+F1 combination is the shortcut key for this mode.

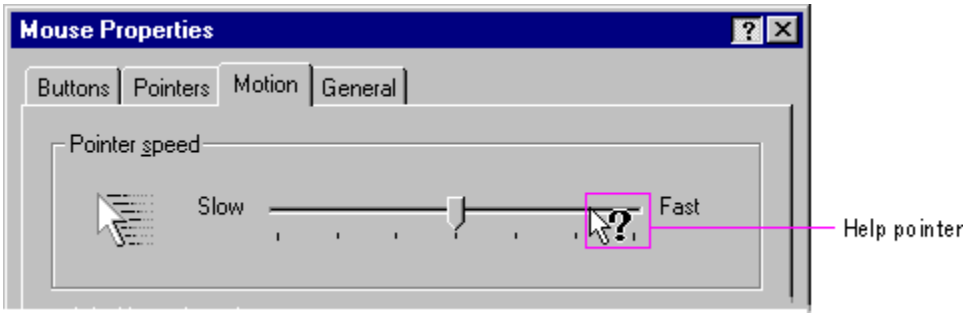



**Figure 12.2 A context-sensitive Help pointer**

Display the context-sensitive Help pointer only over the window that provides context-sensitive Help; that is, only over the active window from which the What's This? command was chosen.

In this mode, when the user clicks an object with mouse button 1 (for pens, tapping), display a context-sensitive Help pop-up window for that object. The context-sensitive Help window provides a brief explanation about the object and how to use it, as shown in Figure 12.3. Once the context-sensitive Help window is displayed, return the pointer and pointer operation to its usual state.

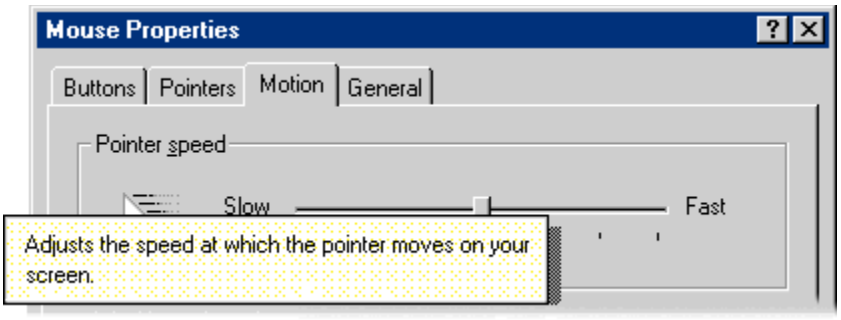



**Figure 12.3 A pop-up window for context-sensitive Help**

If the user presses a shortcut key that applies to a window that is in contextual Help mode, you can display a contextual Help pop-up window for the command associated with that shortcut key.

However, there are some exceptions to this interaction. First, if the user chooses a menu title, either in the menu bar or a cascading menu, maintain the mode until the user chooses a menu item and then display the context-sensitive Help window. Second, if the user clicks the item with mouse button 2 and the object supports a pop-up menu, maintain the mode until the user chooses a menu item or cancels the menu. If the object does not support a pop-up menu, the interaction should be the same as clicking it with mouse button 1. Finally, if the chosen object or location does not support context-sensitive Help or is otherwise an inappropriate target for context-sensitive Help, cancel the context-sensitive Help mode.

If the user chooses the What's This? command a second time, clicks outside the window, or presses the ESC key, cancel the context-sensitive Help mode. Restore the pointer to its usual image and operation in that context.

When the user chooses the What's This? command from a pop-up menu (as shown in Figure 12.4), the interaction is slightly different. Because the user has identified the object by clicking mouse button 2, there is no need for entering the context-sensitive Help mode. Instead, immediately display the context-sensitive Help pop-up window for that object.

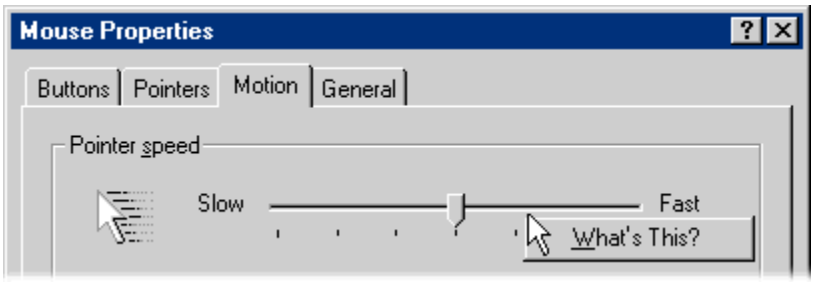



**Figure 12.4 A pop-up menu for a control**

The F1 key is the shortcut key for this form of interaction; that is, pressing F1 displays a context-sensitive Help window for the object that has the input focus.

## Guidelines for Writing Context-Sensitive Help

When authoring context-sensitive Help information, you are answering the question “What is this?” Indicate the action associated with the item. In English versions, begin the description with a verb; for example, “Adjusts the speed of your mouse,” or “Provides a place for you to type in a name for your document.” For command buttons, you may use an imperative form — for example, “Click this to close the window.” When describing a function or object, use words that explain the function or object in common terms instead of technical terminology or jargon. For example, instead of “Undoes the last action,” say “Reverses the last action.”

In the explanation, you might want to include “why” information. You can also include “how to” information, but if the procedure requires multiple steps, consider supporting this information using task-oriented Help. Keep your information brief, but as complete as possible so that the Help window is easy and quick to read.

As an option, you can provide context-sensitive Help information for your supported file types by registering a What’s This? command for the type, as shown in Figure 12.5. This allows the user to choose the “What’s This?” command from the file icon’s pop-up menu to get information about an icon representing that type. When defining this Help information, include the type name and a brief description of its function, using the previously described guidelines.

For more information about registering commands for file types and about type names, see Integrating with the System.

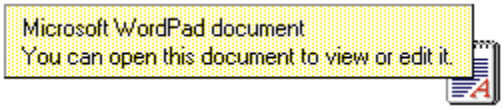



**Figure 12.5 Context-sensitive Help information for an icon**

## Tooltips

Another form of contextual user assistance are tooltips. *Tooltips* are small pop-up windows that display the name of a control when the control has no text label. The most common use of tooltips is for toolbar buttons that have graphic labels, as shown in Figure 12.6, but they can be used for any control.


**Figure 12.6 A tooltip for a toolbar button**

Display a tooltip after the pointer, or pointing device, remains over the button for a short period of time. The tooltip remains displayed until the user presses the button or moves off of the control, or after another time-out. If the user moves the pointer directly to another control supporting a tooltip, ignore the time-out and display the new tooltip immediately, replacing the former one.

For more information about toolbars and tooltip controls, see Menus, Controls, and Toolbars.

If you use the standard toolbar control, the system automatically provides support for tootips. It also includes a tooltip control that can be used in other contexts. If you create your own tooltip controls, make them consistent with the system-supplied controls.

## Status Bar Messages

You can also use a status bar to provide contextual user assistance. However, if you support the user's choice of displaying a status bar, avoid using it for displaying information or access to functions that are essential to basic operation and not provided elsewhere in the application's interface. In addition, because the status bar's location may not be near the user area of activity, the user may not always notice a status bar message. As a result, it is best to consider status bar messages as a secondary or supplemental form of user assistance.

In addition to displaying state information about the context of the activity in the window, you can display descriptive messages about menu and toolbar buttons, as shown in Figure 12.7. Like tooltips, the window typically must be active to support these messages. When the user moves the pointer over a toolbar button or presses the mouse button on a menu or button, display a short message the describing use of the associated command.

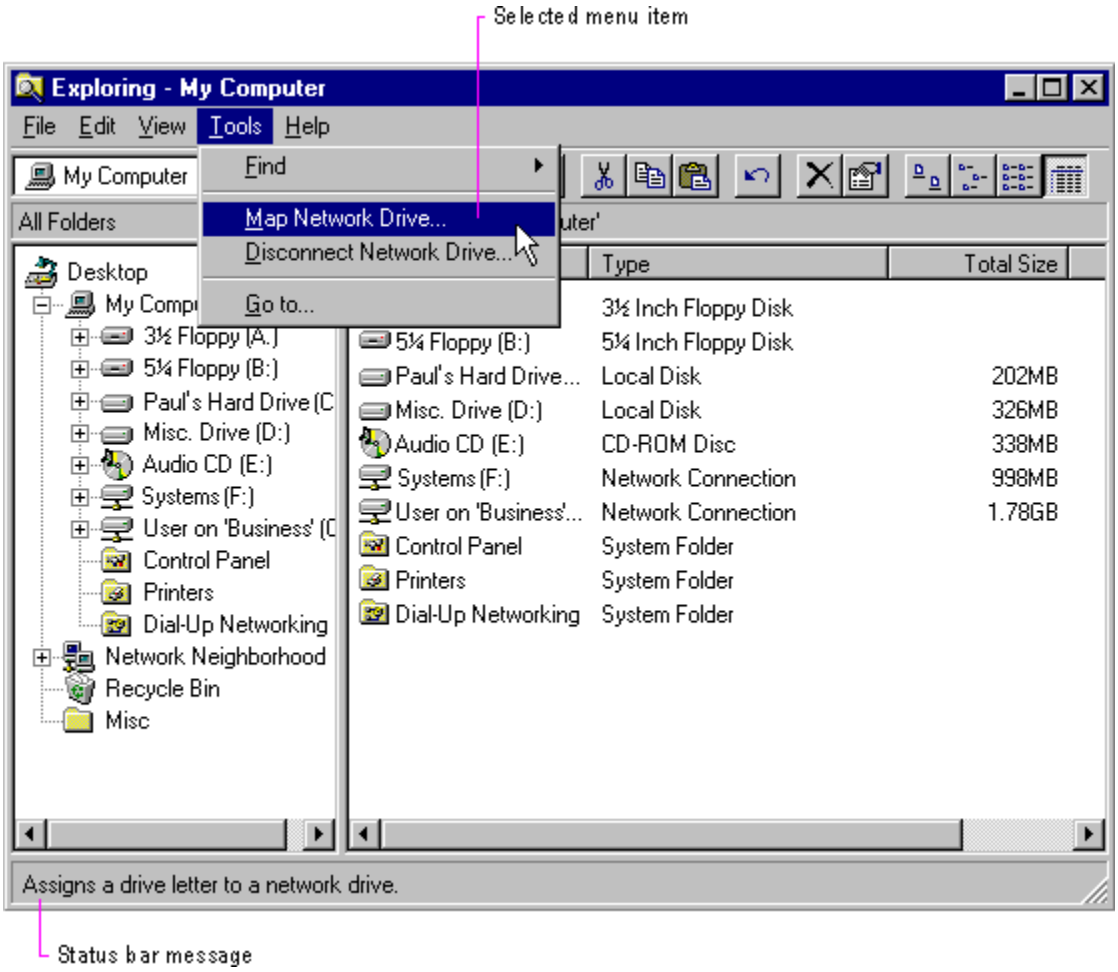



**Figure 12.7 A status bar message for a selected menu command**

A status bar message can include a progress indicator control or other forms of feedback about an ongoing process, such as printing or saving a file, that the user initiated in the window. Although you can display progress information in a message box, you may want to use the status bar for background processes so that the window's interface is not obscured by the message box.

# Guidelines for Writing Status Bar Messages

When writing status bar messages, begin the text with a verb in the present tense and use familiar terms — avoiding jargon. For example, say “Cuts the selection and puts it on the Clipboard.” Try to be as brief as possible so the text can be easily read, but avoid truncation.

Be constructive, not just descriptive, informing the user about the purpose of the command. When describing a command with a specific function, use words specific to the command. If the scope of the command has multiple functions, try to summarize. For example, say “Contains commands for editing and formatting your document.”

When defining messages for your menu and toolbar buttons, don’t forget their unavailable, or disabled, state. Provide an appropriate message to explain why the item is not currently available. For example, when the user selects a disabled Cut command you could display “This command is not available because no text is selected.”

## The Help Command Button

You can also provide contextual Help for a property sheet, dialog box, or message box by including a Help button in that window, as shown in Figure 12.8. When the user chooses the Help command button, display the Help information in a Help secondary window, rather than a context-sensitive Help pop-up window.

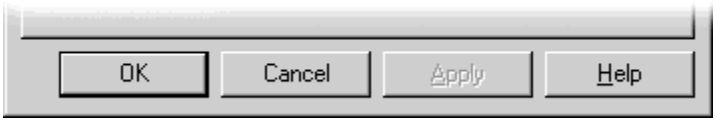



**Figure 12.8 A Help button in a secondary window**

The user assistance provided by a Help command button differs from the “What’s This?” form of Help. Command button Help should provide an overview, summary assistance, or explanatory information for that window. For example, for a message box, it can provide more information about causes and remedies for the reason the message was displayed. Consider the Help command an optional, secondary form of contextual user assistance, not a substitute for context-sensitive, “What’s This?” Help. Don’t use it as a substitute for clear, understandable designs for your secondary windows.

# Task-Oriented Help

*Task-oriented help* provides the steps for carrying out a task. It can involve a number of procedures. You present task-oriented Help in task Help topic windows.

## Task Topic Windows

Task Help topic windows are displayed as primary windows. The user can size this window like any other primary window.

You provide primary access to task Help topics through the Help Topics browser, described in a later topic. You can also include access to specific topics through other interfaces, such as navigation links placed in other Help topics.

The window style is referred to as a primary window because of its appearance and operation. In technical documentation, this window style is sometimes referred to as a Help secondary window.

Task topic windows include a set of command buttons at the top of the window (as shown in Figure 12.9) that provide the user access to the Help Topics browser, the previously selected topic, and other Help commands, such as copying and printing a topic. You can define which buttons appear by defining them in your Help files.

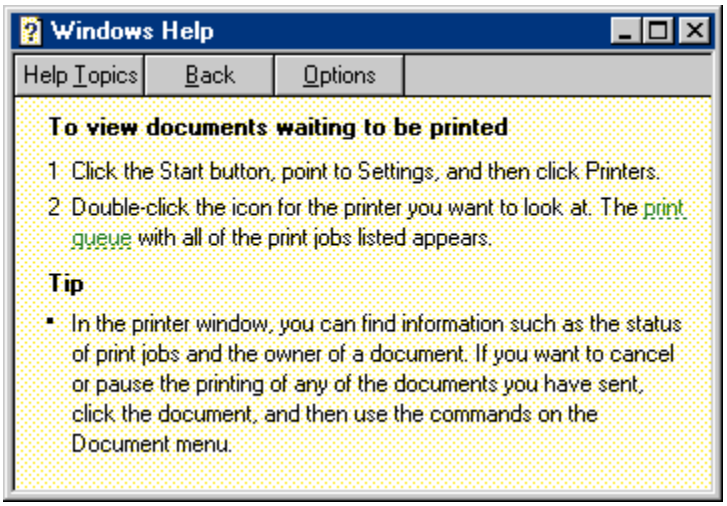



**Figure 12.9 A window for a task Help topic**

Although you can define the size and location of a task Help topic window to the specific requirements of your application, it is best to size and position the window so as to cover the minimum of space, but make it large enough to allow the user to read the topic, preferably without having to scroll the window. This makes it easier for the novice user who may be unfamiliar with scrolling.

The title bar text of the Help topic identifies the context supplying the topic window. Consider naming the Help file to also match. Include a topic title as part of the body of the topic. Define the topic title to correspond to the entries you include in the Help Topics browser that provide direct access to the topic. This does not mean that you must use the same wording as those entries, but they should be similar enough to allow the user to recognize their relationship.

Like tooltips, the default interior color of a task topic window should use the system color setting for Help windows. This allows the user to more easily distinguish the Help topic from their other windows. However, for specialized topics, you can set the color of a task topic window.

## Guidelines for Writing Task Help Topics

The buttons that appear at the top of a task Help topic window are defined by your Help file. At a minimum, you should provide a button that displays the Help Topics browser dialog box, a Back button to return the user to the previous topic, and buttons that provide access to other functions, such as Copy and Print.

To provide access to the Help Topics browser dialog box, include a Help Topics button. This displays the Help Topics browser window on the tabbed page that the user was viewing when the window was last displayed. Although this is the most common form of access to the Help Topics browser window, alternatively you can include buttons, such as Contents and Index, that correspond to the tabbed pages to provide the user with direct access to those pages when the dialog box is displayed.

As with context-sensitive Help, when writing task Help information topics, make them complete, but brief. However, in task Help topics, focus on “how” information rather than “what” or “why.” Task Help should assist the user in completing a task, not try to document everything there is to know about a topic. If there are multiple alternatives, pick one method — usually the simplest, most common method for a specific procedure. If you want to include information on alternative methods, provide access to them through other choices or commands.

If you keep the procedure to four or fewer steps, the user will not need to scroll the window. Avoid introductory, conceptual, or reference material in the procedure.

Also, take advantage of the context of a procedure. For example, if a property sheet includes a slider control that is labeled “Slow” at one end and “Fast” at the other, be concise. Say “Move the slider to adjust the speed” instead of “To increase the speed, move the slider to the right. To decrease the speed, move the slider to the left.” If you refer to a control by its label, capitalize each word in the label, even though the label has only the first word capitalized. This helps distinguish the label from the rest of your text.

Optionally, you can include a Related Topics button in your topic window to provide access to other topics. When the user chooses this button, display the Topics Found dialog box (as shown in Figure 12.14).

## Shortcut Buttons

Task Help topic windows can also include a shortcut or “do it” button that provides the user with a shortcut or automated form of performing a particular step, as shown in Figure 12.10. For example, use this to automatically open a particular dialog box, property sheet, or other object so that the user does not have to search for it.

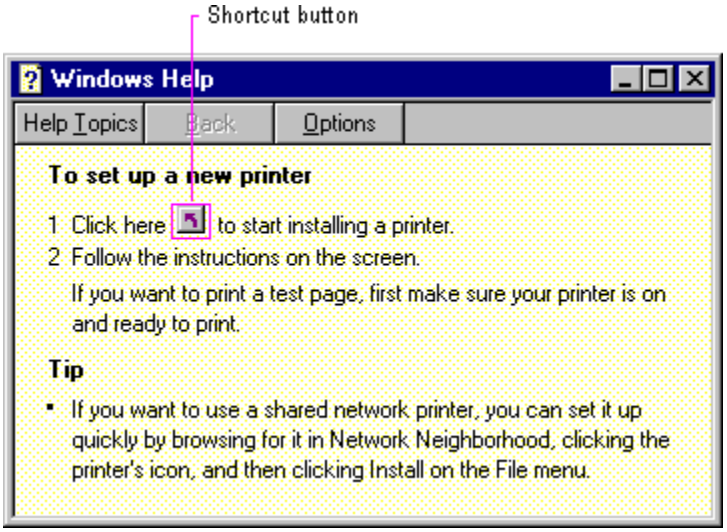



**Figure 12.10 A task Help topic with a shortcut button**

Shortcut buttons not only provide efficiency for the user, they also reduce the amount of information you may need to present and the user needs to read. However, you need not use the buttons as a substitute for doing the task or a specific step in the task; particularly if you want to support the user being able to accomplish the task without using Help. For common tasks, you may want a balance, including information that tells the user how to do the task, and shortcut buttons that make stepping through the task easier. For example, you might include text that reads “Click here to display the Display properties” and a shortcut button.

## Reference Help

*Reference Help* is a form of Help information that serves more as online documentation. Use reference Help to document the features of a product or as a user's guide to a product. Often the use determines the balance of text and graphics used in the Help file. Reference-oriented documentation typically includes more text and follows a consistent presentation of information. User's guide documentation typically organizes information by specific tasks and may include more illustrations.

## The Reference Help Window

When designing reference Help, use a Help primary window style (sometimes called a “main” Help window), as shown in Figure 12.11, rather than the context-sensitive Help pop-up windows or task Help topic windows.

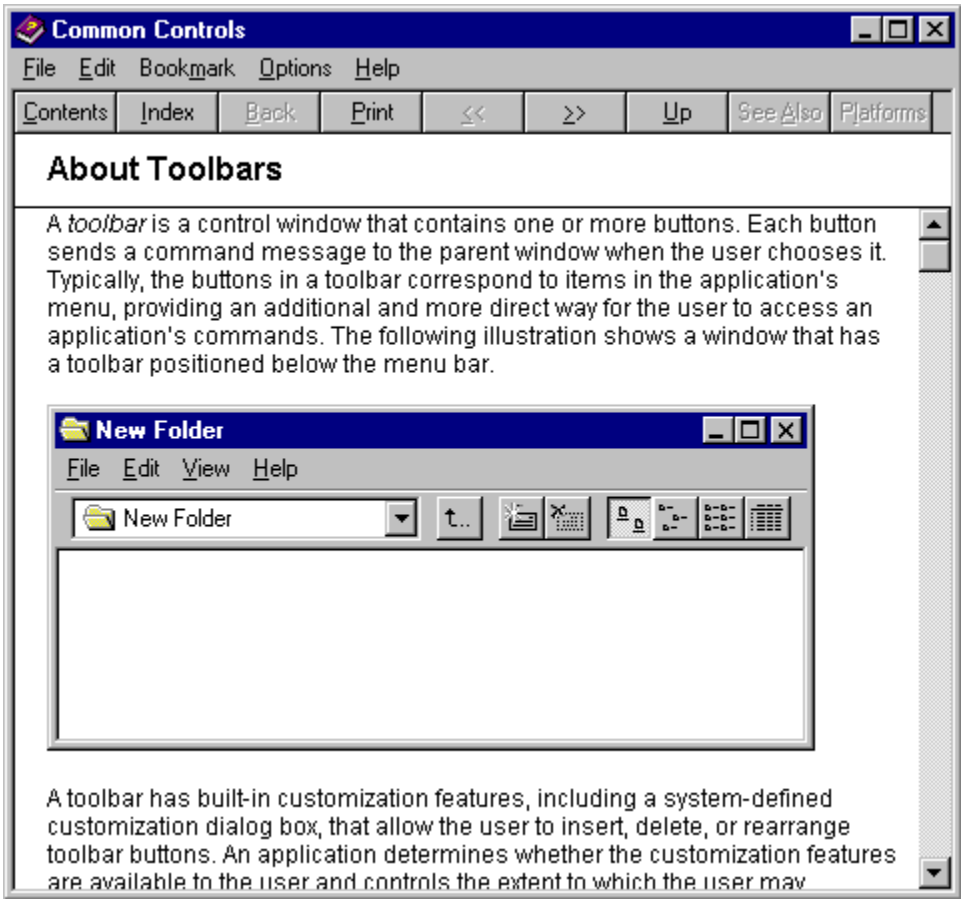



**Figure 12.11 A reference Help window**

You can provide access to reference Help in a variety of ways. The most common is as an explicit menu command in the Help drop-down menu, but you can also provide access using a toolbar button, or even as a specific file object (icon).

A reference Help window includes a menu bar, with File, Edit, Bookmark, Options, and Help entries and a toolbar with Contents, Index, Back, and Print buttons. The system provides these features by default for a “main” Help window. These features support user functions, such as opening a specific Help file (using the Help Topics

browser), copying and printing topics, creating annotations and bookmarks for specific topics, and setting the Help window’s properties. You can add other buttons to this window to tailor your online documentation to fit your particular user needs.

Although the reference Help style can provide information similar to that provided in contextual Help and task Help, these forms of Help are not exclusive of each other. Often the combination of all these items provides the best solution for user assistance. They can also be supplemented with other forms of user assistance.

# Guidelines for Writing Reference Help

Reference Help topics can include text, graphics, animations, video, and audio effects. Follow the guidelines included throughout this guide for recommendations on using these elements in the presentation of information. In addition, the system provides some special support for Help topics.

## Adding Menus and Toolbar Buttons

You can author additional menus and buttons to appear in the reference Help window. However, you cannot remove existing menus.

Because reference Help files typically include related topics, include Previous Topic and Next Topic browse buttons in your Help window toolbar. Another common button you may want to include is a See Also button that either displays a pop-up window or the Topics Found dialog box (as shown in Figure 12.14) with the related topics. Other common buttons include Up for moving to the parent or overview topic and History to display a list of the topics the user has viewed so they can return directly to a particular topic.

Make toolbar buttons contextual to the topic the user is viewing. For example, if the current topic is the last in the browse chain, disable the Next Topic button. When deciding whether to disable or remove a button, follow the guidelines defined in this guide for menus.

## Topic Titles

Always provide a title for the current topic. The title identifies the topic and provides the user with a landmark within the Help system. The title should correspond to the entries you include in the Help Topics browser window. Use the title bar text of the window to identify the context and supplier of the topic. The Help filename should also match.

## Nonscrolling Regions

If your topics are very long, you may want to include a nonscrollable region in your Help file. A nonscrolling region allows you to keep the topic title and other information visible when the user scrolls. A nonscrolling region appears with a line at its bottom edge to delineate it from the scrollable area. Display the scroll bar for the scrollable area of the topic so that its top appears below the nonscrolling region, not overlapped within that region.

## Jumps

A jump is a button or interactive area that triggers an event when the user clicks on it. You can use a jump as a one-way navigation link from one topic to another, either within the same topic window, to another topic window, or a topic in another Help file.

You can also use jumps to display a pop-up window. As with pop-up windows for context-sensitive Help, use this form of interaction to support a definition or explanatory information about the word or object that the user clicks.

Jumps can also carry out particular commands. Shortcut buttons used in Help task topics are this form of a jump.

You need to provide visual indications to distinguish a jump from noninteractive areas of the window. You can do this by displaying a jump as a button, changing the pointer image to indicate an interactive element, formatting the item with some other visual distinction such as color or font, or a combination of these methods. The default system presentation for text jumps is green underlined text.

# The Help Topics Browser

The Help Topics browser dialog box provides user access to Help information. To open this window, include a Help Topics menu item on the Help drop-down menu. Alternatively, you can include menu commands that open the window to a particular tabbed page — for example, Contents, Index, and Find Topic.

In addition, provide a Help Topics button in the toolbar of a task or reference Help topic window. When the user chooses this button, display the Help Topics browser window with the last page the user accessed. If you prefer, provide Contents, Index, and Find Topic buttons for direct access to a specific page. For example, by default, reference Help windows include Contents and Index button access to the Help Topics browser.

## The Help Topics Tabs

Opening the Help Topics window displays a set of tabbed pages. The default pages include Contents, Index, and Find tabs. You can author additional tabs.

## The Contents Page

The Contents page displays the list of topics organized by category, as shown in Figure 12.12. A book icon represents a category or group of related topics, and a page icon represents an individual topic. You can nest topic levels, but avoid more than three levels, as this can make access cumbersome.

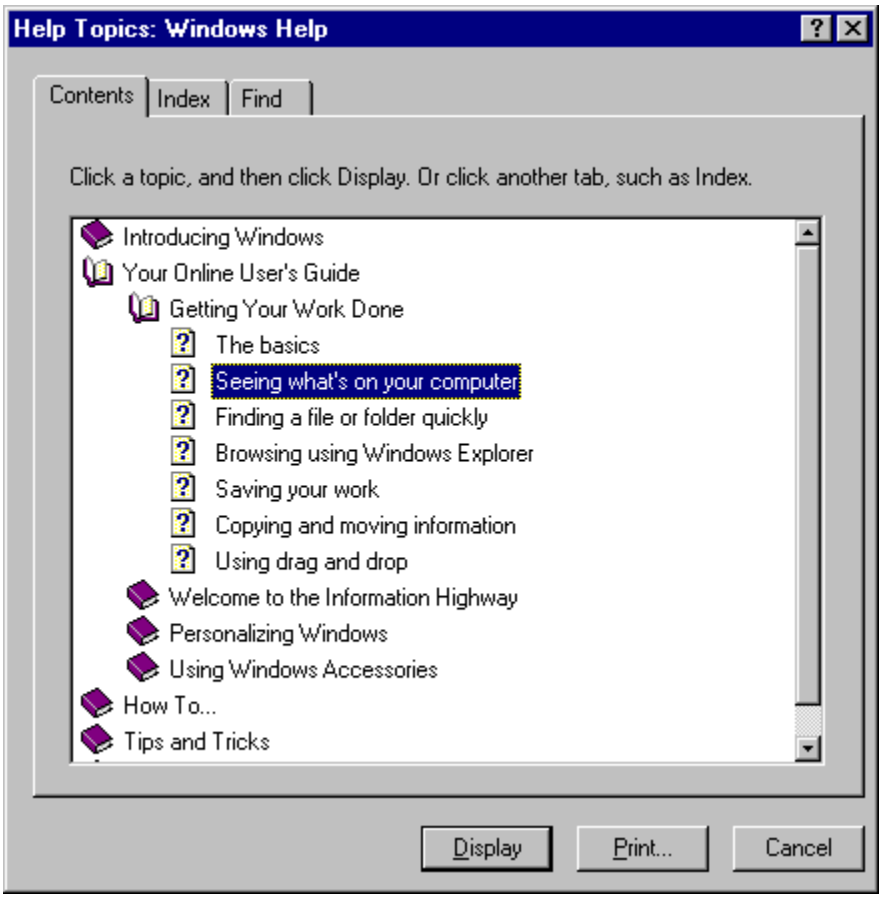



**Figure 12.12 The Contents page of the Help topics browser**

The buttons at the bottom of the page allow the user to open or close a “book” of topics and display a particular topic. The Print button prints either a “book” of topics or a specific topic depending on which the user selects. The outline also supports direct interaction, such as double-clicking, for opening the outline or a topic.

# Guidelines for Writing Help Contents Entries

The entries listed on the Contents page are based on what you author in your Help files. Define them to allow the user to see the organizational relationship between topics. Make the topic titles you include for your software brief, but descriptive, and correspond to the actual topic titles.

## The Index Page

The Index page of the browser organizes the topics by keywords that you define for your topics, as shown in Figure 12.13.

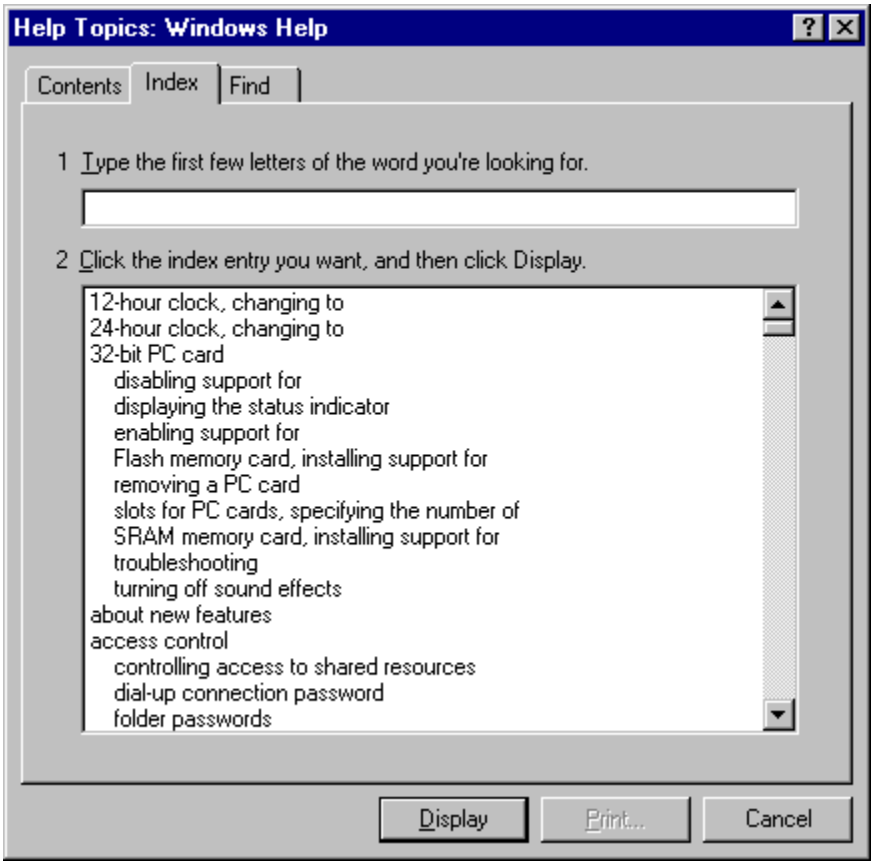



**Figure 12.13 The Index page of the Help Topics browser**

The user can enter a keyword or select one from the list. Choosing the Display button displays the topic associated with that keyword. If there are multiple topics that use the same keyword, then another secondary window is displayed that allows the user to choose from that set of topics, as shown in Figure 12.14. You can also use this dialog box to provide access to related topics by including a See Also button or Related Topics button in a topic window.

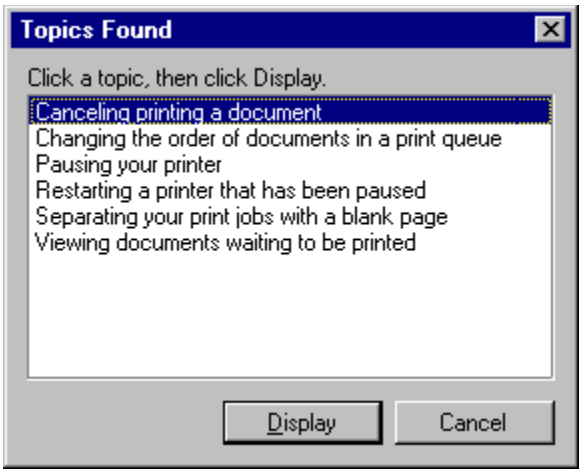



**Figure 12.14 The Help topics window**

# Guidelines for Writing Help Index Keywords

Provide an effective keyword list to help users find the information they are looking for. When deciding what keywords to provide for your topics, consider the following categories:

- Words for a novice user.
- Words for an advanced user.
- Common synonyms of the words in the keyword list.
- Words that describe the topic generally.
- Words that describe the topic discretely.

## The Find Page

The Find page, as shown in Figure 12.15, provides full-text search functionality that allows the user to search for any word or phrase in the Help file. This capability requires a full-text index file, which you can create when building the Help file, or which the user can create when using the Find page.

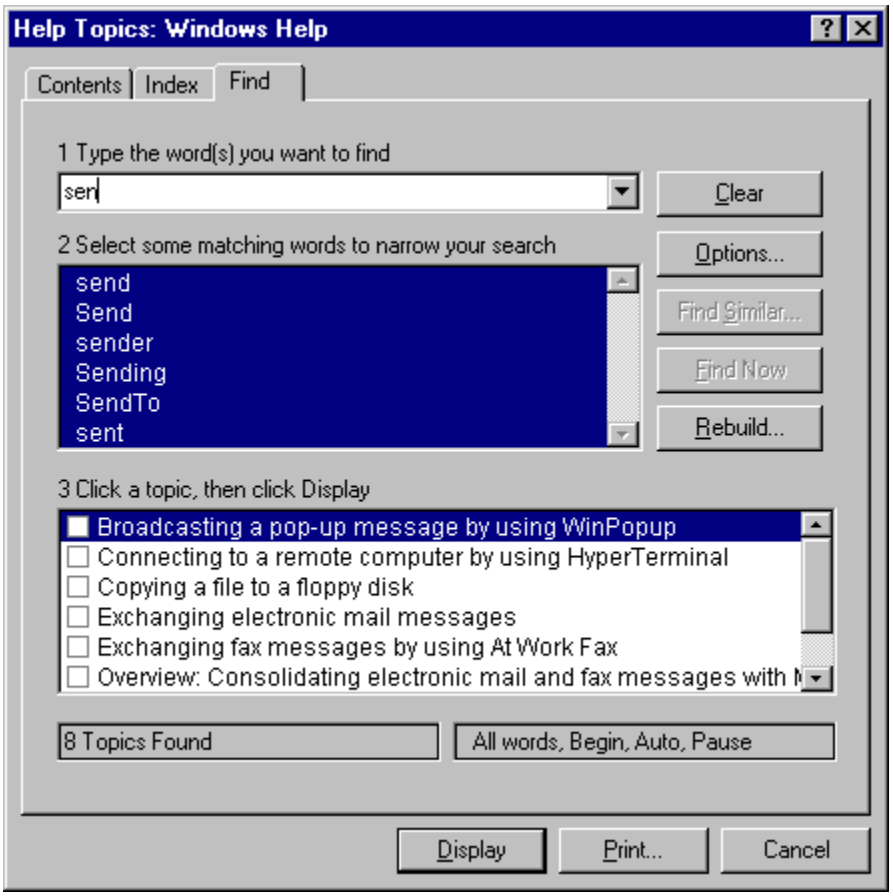



**Figure 12.15 The Find page of the Help Topics browser**

# Wizards

A *wizard* is a special form of user assistance that automates a task through a dialog with the user. Wizards help the user accomplish tasks that can be complex and require experience. Wizards can automate almost any task, including creating new objects and formatting the presentation of a set of objects, such as a table or paragraph. They are especially useful for complex or infrequent tasks that the user may have difficulty learning or doing.

The system provides support for creating a wizard using the standard property sheet control. For more information about this control, see Menus, Controls, and Toolbars.

However, wizards are not well-suited to teach a user how to do something. Although wizards assist the user in accomplishing a task, they should be designed to hide many of the steps and much of the complexity of a given task. Similarly, wizards are not intended to be used for tutorials; wizards should operate on real data. For instructional user assistance, consider task Help or tutorial-style interfaces.

Do not rely on wizards as a solution for ineffective designs; if the user relies on a wizard too much it may be an indication of an overly complicated interface, not good wizard design. In addition, consider using a wizard to supplement, rather than replace, the user's direct ability to perform a specific task. Unless the task is fairly simple or done infrequently, experienced users may find a wizard to be inefficient or not provide them with sufficient access to all functionality.

Wizards may not always appear as an explicit part of the Help interface. You can provide access to them in a variety of ways, including toolbar buttons or even specific icons, such as templates.

For more information about templates, see General Interaction Techniques.

# Guidelines for Designing Wizards

A wizard is a series of presentations or pages, displayed in a secondary window, that helps the user through a task. The pages include controls that you define to gather input from the user; that input is then used to complete the task for the user.

Optionally, you can define wizards as a series of secondary windows through which the user navigates. However, this can lead to increased modality and screen clutter, so using a single secondary window is recommended.

At the bottom of the window, include the following command buttons that allow the user to navigate through the wizard.

| Command | Action |
|---|---|
| < Back | Returns to the previous page. (Remove or disable the button on the first page.) |
| Next > | Moves to the next page in the sequence, maintaining whatever settings the user provides in previous pages. |
| Finish | Applies user-supplied or default settings from all pages and completes the task. |
| Cancel | Discards any user-supplied settings, terminates the process, and closes the wizard window. |

Use the title bar text of the wizard window to clearly identify the purpose of the wizard. But, because wizards are secondary windows, they should not appear in the taskbar. You may optionally include a context-sensitive What's This? Help title bar button as well.

On the first page of a wizard, include a graphic in the left side of the window, as shown in Figure 12.16. The purpose of this graphic is to establish a reference point, or theme — such as a conceptual rendering, a snapshot of the area of the display that will be affected, or a preview of the result. On the top right portion of the wizard window, provide a short paragraph that welcomes the user to the wizard and explains what it does. You may also include controls for entering or editing input to be used by the wizard, if there is sufficient space. However, avoid offering too many choices or choices that may be difficult to distinguish or understand without context.

**Figure 12.16 An introductory page of a wizard**

On subsequent pages you can continue to include a graphic for consistency or, if space is critical, use the entire width of the window for displaying instructional text and controls for user input. When using graphics, include pictures that help illustrate the process, as shown in Figure 12.17. Include default values or settings for all controls where possible.

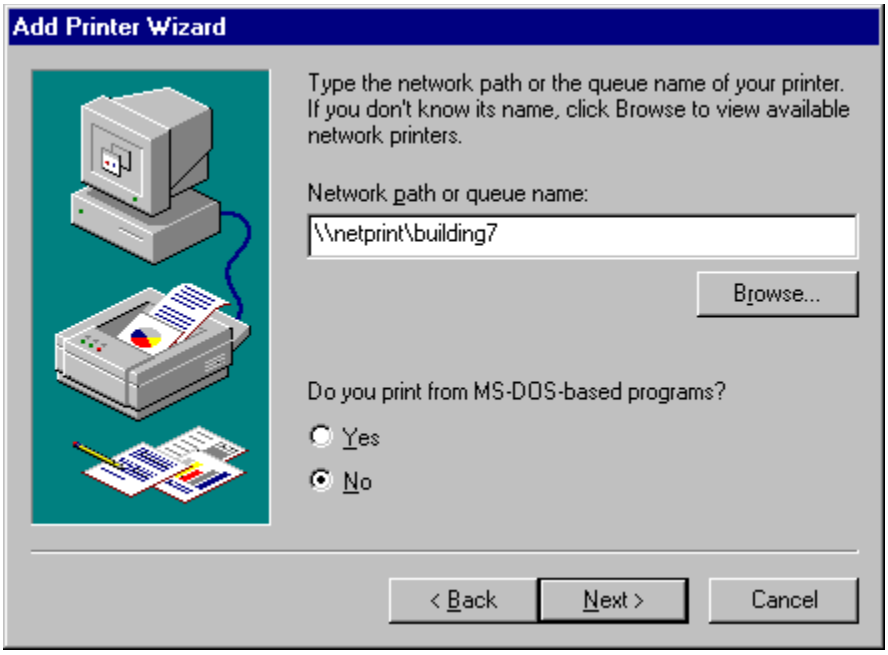



**Figure 12.17 Input page for a wizard**

You can include the Finish button at any point that the wizard can complete the task. For example, if you can provide reasonable defaults, you can even include the Finish button on the first page. Place the Finish button to the right and adjacent to the Next button. This allows the user to step through the entire wizard or only the page on which they wish to provide input. Otherwise, if the user needs to step through each page of the wizard, replace the Next button with the Finish button on the last page of the wizard. Also on the last page of the wizard, indicate to the user that the wizard is prepared to complete the task and instruct the user to click the Finish button.

Design your wizard pages to be easy to understand. It is important that users immediately understand what a wizard is about so they don't feel like they have to read it very carefully to understand what they have to answer. It is better to have a greater number of simple pages with fewer choices than a smaller number of complex pages with too many options or text. In addition, follow the conventions outlined in this guide and consider the following guidelines when designing a wizard:

- Minimize the number of pages that require the display of a secondary window. Novice users are often confused by the additional complexity of secondary windows.
- Avoid a wizard design that requires the user to leave the wizard to complete a task. Less experienced users are often the primary users of a wizard. Asking them to leave the wizard to perform a function can make them lose their context. Instead, design your wizard so that the user can do everything from within the wizard.
- Make it visually clear that user-interface elements that are part of a graphic illustration on a wizard page are not interactive. You can do this by varying the graphic from their normal sizes or rendering them in a more abstract representation.
- Avoid advancing pages automatically. The user may not be able to read the information before a page advances. In addition, wizards are intended to allow the user to be in control of the process that the wizard automates.
- Display a wizard window so that the user can recognize it as the primary point of input. For example,

if you display a wizard from a control the user chooses in a secondary window, you may need to place the wizard window so that it partially obscures that secondary window.

- Make certain that the design alternatives offered by your wizard provide the user with positive results. You can use the context, such as the selection, to determine what options may be reasonable to provide.
- Make certain that it is obvious how the user can proceed when the wizard has completed its process. This may be accomplished by the text you include on the last page of the wizard.

# Guidelines for Writing Text for Wizard Pages

Use a conversational, rather than instructional, writing style for the text you provide on the screens. The following guidelines can be used to assist you in writing the textual information:

- Use words like “you” and “your.”
- Start most questions with phrases like “Which option do you want...” or “Would you like....” Users respond better to questions that enable them to do a task than being told what to do. For example, “Which layout do you want?” works better in wizards than “Choose a layout.”
- Use contractions and short, common words. In some cases, it may be acceptable to use slang, but you must consider localization when doing so.

For more information about localization design, see Special Design Considerations.

- Avoid using technical terminology that may be confusing to a novice user.
- Try to use as few words as possible. For example, the question “Which style do you want for this newsletter?” could be written simply as “Which style do you want?”
- Keep the writing clear, concise, and simple, but remember not to be condescending.

# Introduction

What we see influences how we feel and what we understand. Visual information communicates nonverbally, but very powerfully. It can include cues that motivate, direct, or distract. This topic covers the visual and graphic design principles and guidelines that you can apply to the interface design of your Microsoft Windows-based applications.

 Animation

## Visual Communication

Effective visual design serves a greater purpose than decoration; it is an important tool for communication. How you organize information on the screen can make the difference between a design that communicates a message and one that leaves a user feeling puzzled or overwhelmed.

Even the best product functionality can suffer if its visual presentation does not communicate effectively. If you are not trained in visual or information design, it is a good idea to work with a designer who has education and experience in this field and include that person as a member of the design team early in the development process. Good graphic designers provide a perspective on how to take the best advantage of the screen and how to use effectively the concepts of shape, color, contrast, focus, and composition. Moreover, graphic designers understand how to design and organize information, and the effects of fonts and color on perception.

## Composition and Organization

We organize what we read and how we think about information by grouping it spatially. We read a screen in the same way we read other forms of information. The eye is always attracted to the colored elements before black and white, to isolated elements before elements in a group, and to graphics before text. We even read text by scanning the shapes of groups of letters. Consider the following principles when designing the organization and composition of visual elements of your interface: hierarchy of information, focus and emphasis, structure and balance, relationship of elements, readability and flow, and unity of integration.

# Hierarchy of Information

The principle of hierarchy of information addresses the placement of information based on its relative importance to other visual elements. The outcome of this ordering affects all of the other composition and organization principles, and determines what information a user sees first and what a user is encouraged to do first. To further consider this principle, ask these questions:

- What information is most important to a user?

  In other words, what are the priorities of a user when encountering your application's interface. For example, the most important priority may be to create or find a document.
- What does a user want or need to do first, second, third, and so on?

  Will your ordering of information support or complicate a user's progression through the interface?
- What should a user see on the screen first, second, third, and so on?

  What a user sees first should match the user's priorities when possible, but can be affected by the elements you want to emphasize.

## Focus and Emphasis

The related principle of focus and emphasis guides you in the placement of priority items. Determining focus involves identifying the central idea, or the focal point, for activity. Determine emphasis by choosing the element that must be prominent and isolating it from the other elements or making it stand out in other ways.

Where the user looks first for information is an important consideration in the implementation of this principle. Culture and interface design decisions can govern this principle. People in western cultures, for example, look at the upper left corner of the screen or window for the most important information. So, it makes sense to put a top-priority item there, giving it emphasis.

# Structure and Balance

The principle of structure and balance is one of the most important visual design principles. Without an underlying structure and a balance of visual elements, there is a lack of order and meaning and this affects all other parts of the visual design. More importantly, a lack of structure and balance makes it more difficult for the user to clearly understand the interface.

## Relationship of Elements

The principle of relationship of elements is important in reinforcing the previous principles. The placement of a visual element can help communicate a specific relationship of the elements of which it is a part. For example, if a button in a dialog box affects the content of a list box, there should be a spatial relationship between the two elements. This helps the user to clearly and quickly make the connection just by looking at the placement.

## Readability and Flow

This principle calls for ideas to be communicated directly and simply, with minimal visual interference. Readability and flow can determine the usability of a dialog box or other interface component. When designing the layout of a window, consider the following:

- Could this idea or concept be presented in a simpler manner?
- Can the user easily step through the interface as designed?
- Do all the elements have a reason for being there?

## Unity and Integration

The last principle, unity and integration, reflects how to evaluate a given design in relationship to its larger environment. When an application's interface is visually unified with the general interface of Windows, the user finds it easier to use because it offers a consistent and predictable work environment. To implement this principle, consider the following:

- How do all of the different parts of the screen work together visually?
- How does the visual design of the application relate to the system's interface or other applications with which it is used?

# Color

Color is a very important property in the visual interface. Because color has attractive qualities, use it to identify elements in the interface to which you want to draw the user's attention — for example, the current selection. Color also has an associative aspect; we often assume there is a relationship between items of the same color. Color also carries with it emotional or psychological qualities. For example, colors are often categorized as cool or warm.

When used indiscriminately, color can have a negative or distracting effect. It can affect not only the user's reaction to your software but also productivity, by making it difficult to focus on a task.

In addition, there are a few more things to consider about using color:

- Although you can use color to show relatedness or grouping, associating a color with a particular meaning is not always obvious or easily learned.
- Color is a very subjective property. Everyone has different tastes in color. What is pleasing to you may be distasteful to someone else.
- Some percentage of your users may work with equipment that only supports monochrome presentation.
- Interpretation of color can vary by culture. Even within a single culture, individual associations with color can differ.
- Some percentage of the population may have color-identification problems. For example, about 9 percent of the adult male population have some form of color confusion.

The following sections summarize guidelines for using color: color as a secondary form of information, use of a limited set of colors, allowing the option to change colors.

## Color as a Secondary Form of Information

Use color as an additive, redundant, or enhanced form of information. Avoid relying on color as the only means of expressing a particular value or function. Shape, pattern, location, and text labels are other ways to distinguish information. It is also a good practice to design visuals in black and white or monochrome first, then add color.

## Use of a Limited Set of Colors

Although the human eye can distinguish millions of different colors, using too many usually results in visual clutter and can make it difficult for the user to discern the purpose of the color information. The colors you use should fit their purpose. Muted, subtle, complementary colors are usually better than bright, highly saturated ones, unless you are really looking for a carnival-like appearance where bright colors compete for the user's attention.

Color also affects color. Adjacent or background colors affect the perceived brightness or shade of a particular color. A neutral color (for example, light gray) is often the best background color. Opposite colors, such as red and green, can make it difficult for the eye to focus. Dark colors tend to recede in the visual space, light colors come forward.

# Options to Change Colors

Because color is a subjective, personal preference, allow the user to change colors where possible. For interface elements, Windows provides standard system interfaces and color schemes. If you base your software on these system properties, you can avoid including additional controls, plus your visual elements are more likely to coordinate effectively when the user changes system colors. This is particularly important if you are designing your own controls or screen elements to match the style reflected in the system.

When providing your own interface for changing colors, consider the complexity of the task and skill of the user. It may be more helpful if you provide palettes, or limited sets of colors, that work well together rather than providing the entire spectrum. You can always supplement the palette with an interface that allows the user to add or change a color in the palette.

## Fonts

Fonts have many functions in addition to providing letterforms for reading. Like other visual elements, fonts organize information or create a particular mood. By varying the size and weight of a font, we see text as more or less important and perceive the order in which it should be read.

At conventional resolutions of computer displays, fonts are generally less legible online than on a printed page. Avoid italic and serif fonts; these are often hard to read, especially at low resolutions. Figure 13.1 shows various font choices.

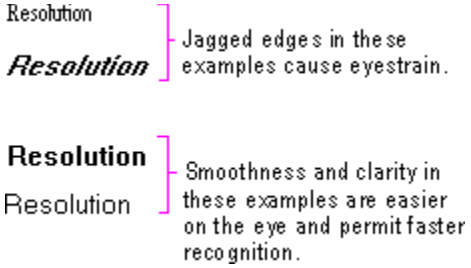



**Figure 13.1 Effective and ineffective font choices**

Limit the number of fonts and styles you use in your software's interface. Using too many fonts usually results in visual clutter.

Wherever possible, use the standard system font for common interface elements. This provides visual consistency between your interface and the system's interface and also makes your interface more easily scaleable. Because many interface elements can be customized by the user, check the system settings for the default system font and set the fonts in your interface accordingly. For more information about system font settings, see Layout.

## Dimensionality

Many elements in the Windows interface use perspective, highlighting, and shading to provide a three-dimensional appearance. This emphasizes function and provides real-world feedback to the user's actions. For example, command buttons have an appearance that provides the user with natural visual cues that help communicate their functionality and differentiate them from other types of information.

Windows bases its three-dimensional effects on a common theoretical light source, the conceptual direction that light would be coming from to produce the lighting and shadow effects used in the interface. The light source in Windows comes from the upper left.

When designing your own visual elements, be careful not to overdo the use of dimensionality. Avoid unnecessary nesting of visual elements and using three-dimensional effects for an element that is not interactive. Introduce only enough detail to provide useful visual cues and use designs that blend well with the system interface.

## Design of Visual Elements

All visual elements influence one another. Effective visual design depends on context. In a graphical user interface, a graphic element and its function are completely interrelated. A graphical interface needs to function intuitively — it needs to look the way it works and work the way it looks.

## Basic Border Styles

Windows provides a unified visual design for building visual components based on the border styles shown in Figure 13.2.

The basic border styles are based on standard system color settings.

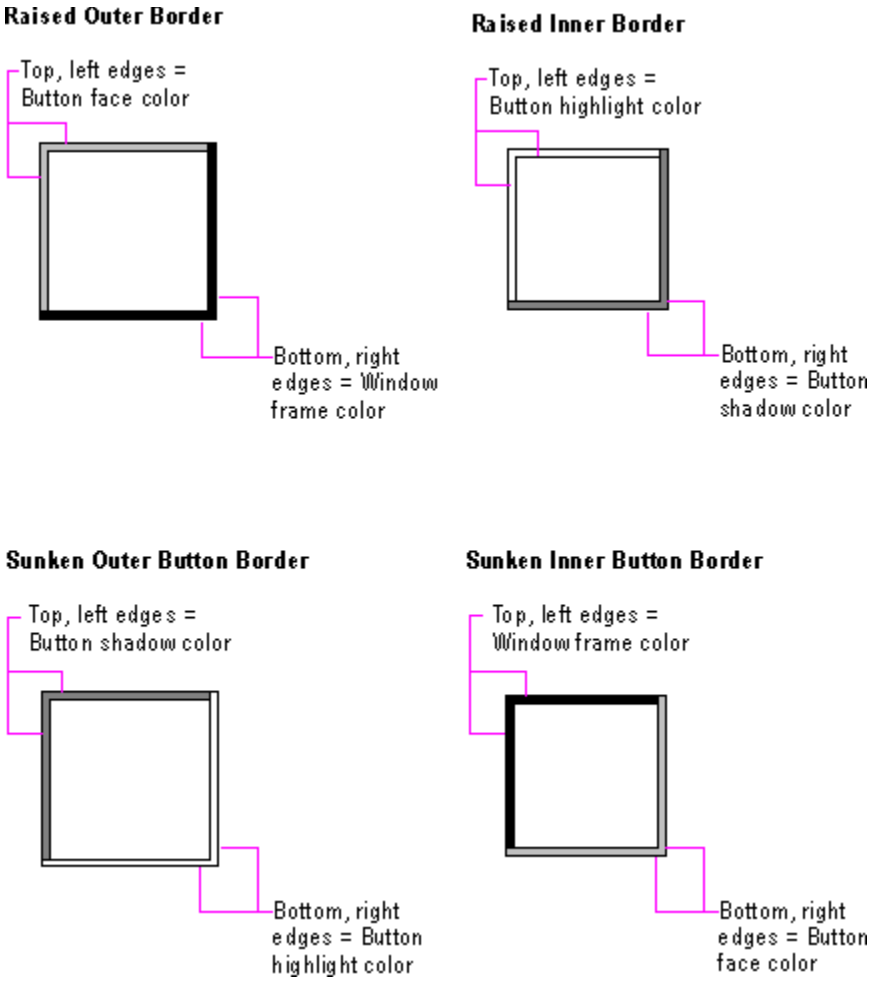



**Figure 13.2 Basic border styles**

| Border style | Description |
|---|---|
| Raised outer border | Uses a single-pixel width line in the button face color for its top and left edges and the window frame color for its bottom and right edges. |
| Raised inner border | Uses a single-pixel width line in the button highlight color for its top and left edges and the button shadow color for its bottom and right edges. |
| Sunken outer border | Uses a single-pixel width line in the button shadow color for its top and left border and the button highlight color for its bottom and right edges. |
| Sunken inner border | Uses a single-pixel width line in the window frame color for its top and left edges and the button face |

color for its bottom and right edges.

The **DrawEdge** function automatically provides these border styles using the correct color settings. For more information about this function, see the documentation included in the Microsoft Win32 Software Development Kit (SDK).

If you use standard Windows controls and windows, these border styles are automatically supplied for your application. If you create your own controls, your application should map the colors of those controls to the appropriate system colors so that the controls fit in the overall design of the interface when the user changes the basic system colors.

## Window Border Style

The borders of primary and secondary windows, except for pop-up windows, use the window border style. Menus, scroll arrow buttons, and other situations where the background color may vary also use this border style. The border style is composed of the raised outer and raised inner basic border styles, as shown in Figure 13.3.

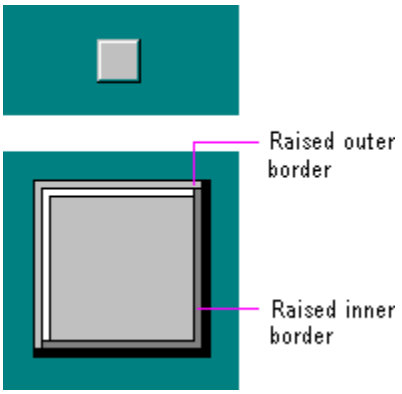



**Figure 13.3 Window border style**

## Button Border Styles

Command buttons use the button border style. The button border style uses a variation of the basic border styles where the colors of the top and left outer and inner borders are swapped when combining the borders, as shown in Figure 13.4.

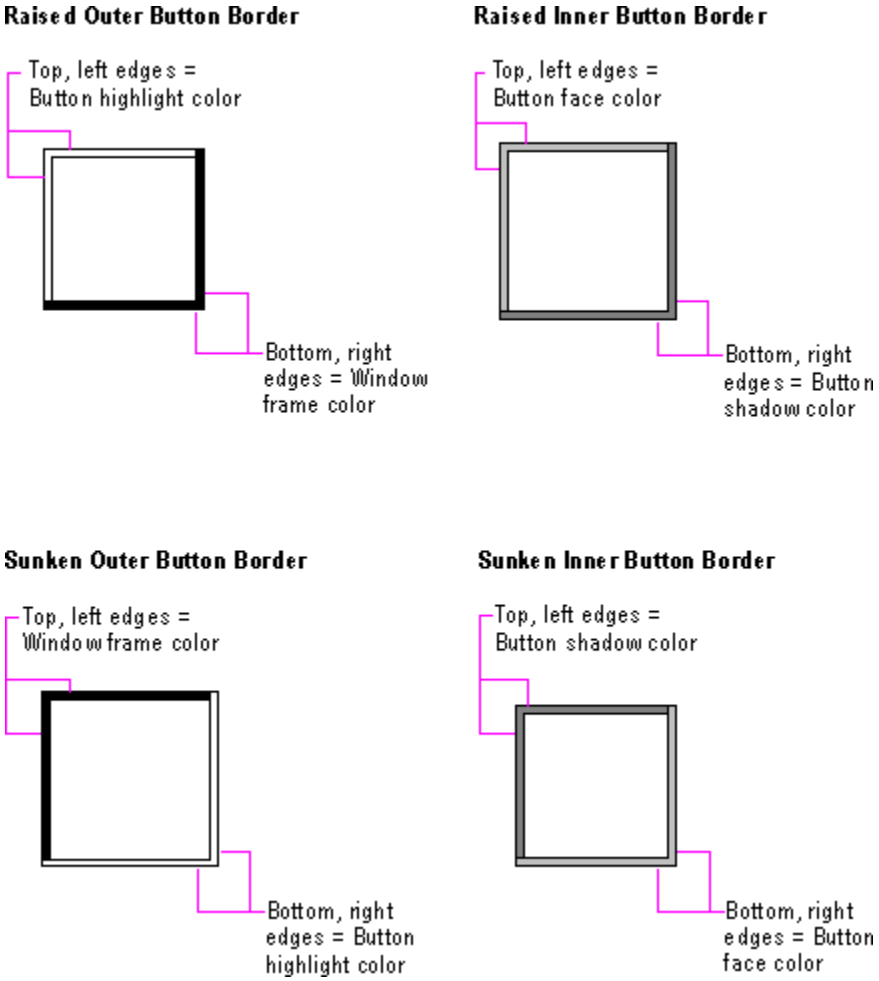



**Figure 13.4 Button border styles**

The normal button appearance combines the raised outer and raised inner button borders. When the user presses the button, the sunken outer and sunken inner button border styles are used, as shown in Figure 13.5.

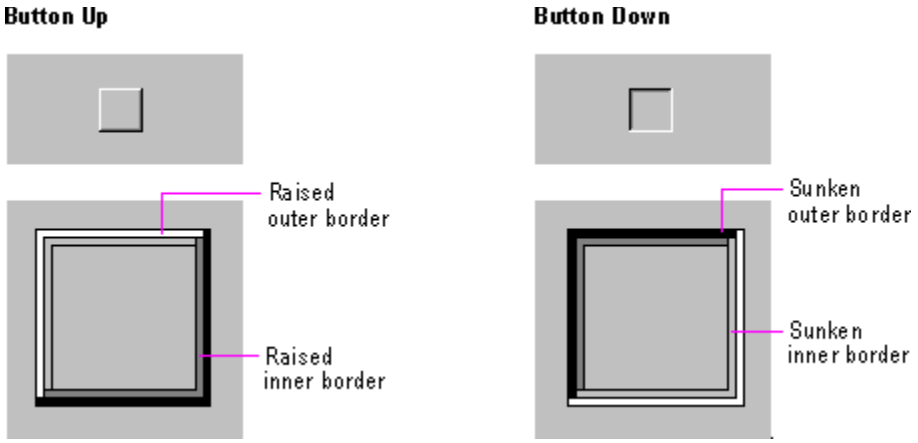



**Figure 13.5 Button up and button down border styles**

## Field Border Style

Text boxes, check boxes, drop-down combo boxes, drop-down list boxes, spin boxes, list boxes, and wells use the field border style, as shown in Figure 13.6. You can also use the style to define the work area within a window. It uses the sunken outer and sunken inner basic border styles.

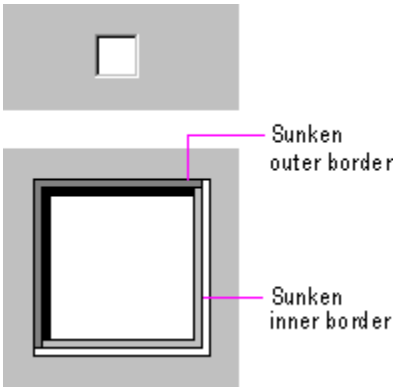



**Figure 13.6 The field border style**

For most controls, the interior of the field uses the button highlight color. However, in wells, the color may vary based on how the field is used or what is placed in the field, such as a pattern or color sample. For text fields, such as text boxes and combo boxes, the interior uses the button face color when the field is read-only or disabled.

## Status Field Border Style

Status fields use the status field border style, as shown in Figure 13.7. This style uses only the sunken outer basic border style.

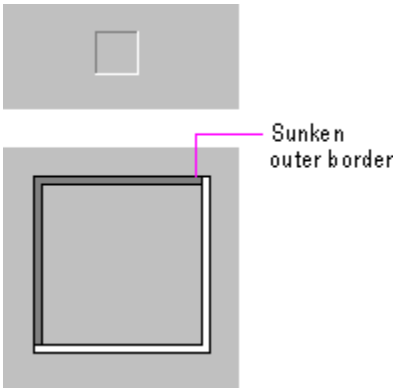



**Figure 13.7 The status field border style**

You use the status field style in status bars and any other read-only fields where the content of the file can dynamically change.

## Grouping Border Style

Group boxes and menu separators use the grouping border style, as shown in Figure 13.8. The style uses the sunken outer and raised inner basic border styles.

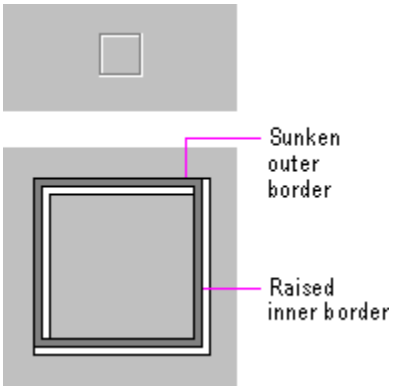



**Figure 13.8 The group border style**

## Visual States for Controls

The visual design of controls includes the various states supported by the control. If you use standard Windows controls, Windows automatically provides specific appearances for these states. If you design your own controls, use the information in the previous section for the appropriate border style and information in the following sections to make your controls consistent with standard Windows controls.

For more information about standard control behavior and appearance, see Menus, Controls, and Toolbars, and the documentation included in the Win32 SDK.

## Pressed Appearance

When the user presses a control, it provides visual feedback on the down transition of the mouse button. (For the pen, the feedback provided is for when the pen touches the input surface and for the keyboard, upon the down transition of the key.)

For standard Windows check boxes and option buttons, the background of the button field is drawn using the button face color, as shown in Figure 13.9.

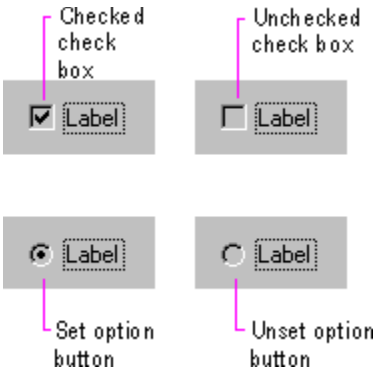



**Figure 13.9 Pressed appearance for check boxes and option buttons**

For command buttons, the button-down border style is used and the button label moves down and to the right by one pixel, as shown in Figure 13.10.

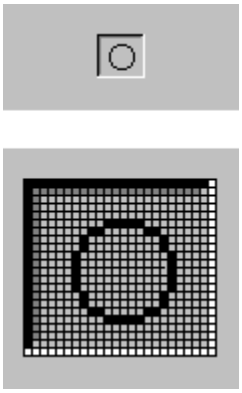


**Figure 13.10 Pressed appearance for a command button**

## Option-Set Appearance

When using buttons to indicate when its associated value or state applies or is currently set, the controls provide an *option-set appearance*. The option-set appearance is used upon the up transition of the mouse button or pen tip, and the down transition of a key. It is visually distinct from the pressed appearance.

Standard check boxes and option buttons provide a special visual indicator when the option corresponding to that control is set. A check box uses a check mark, and an option button uses a dot that appears inside the button, as shown in Figure 13.11.


**Figure 13.11 Option-set appearance for check boxes and option buttons**

When using command buttons to represent properties or other state information, the button face reflects when the option is set. The button continues to use the button-down border style, but a checkerboard pattern (dither) using the color of the button face and button highlight is displayed on the interior background of the button, as shown in Figure 13.12. For configurations that support 256 or more colors, if the button highlight color setting is not white, the button interior background is drawn in a halftone between button highlight color and button face color. The glyph on the button does not otherwise change from the pressed appearance.

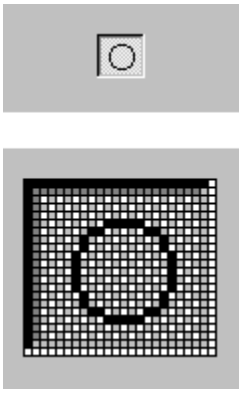


**Figure 13.12 Option-set appearance for a command button**

For well controls (shown in Figure 13.13), when a particular choice is set, place a border around the control, using the window text color and the button highlight color.

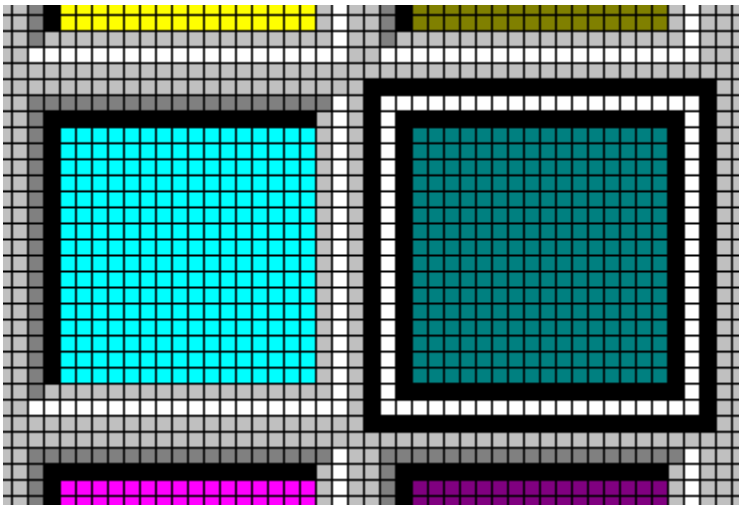


**Figure 13.13 Option-set appearance for a well**

## Mixed-Value Appearance

When a control represents a property or other setting that reflects a set of objects where the values are different, the control is displayed with a *mixed-value* appearance (also referred to as indeterminate appearance), as shown in Figure 13.14.

For most standard controls, leave the field with no indication of a current set value if it represents a mixed value. For example, for a drop-down list, the field is blank.

Standard check boxes support a special appearance for this state that displays the check mark, in the button shadow color, against a checkerboard background that uses the button highlight color and button face color. For configurations that support 256 or more colors, if the button highlight color setting is not white, the interior of the control is drawn in a halftone between button highlight color and button face color.

The system defines the mixed-value states for check boxes as constants BS_3STATE and BS_AUTO3STATE when using the **CreateWindow** and **CreateWindowEx** functions. For more information about these functions, see the documentation included in the Win32 SDK.


**Figure 13.14 Mixed-value appearance for a check box**

For graphical command buttons, such as those used on toolbars, the checkerboard pattern, using the button highlight color and button face color, or the halftone color, is drawn on the background of the button face, as shown in Figure 13.15. The image is converted to a monochrome presentation and drawn in the button shadow color.

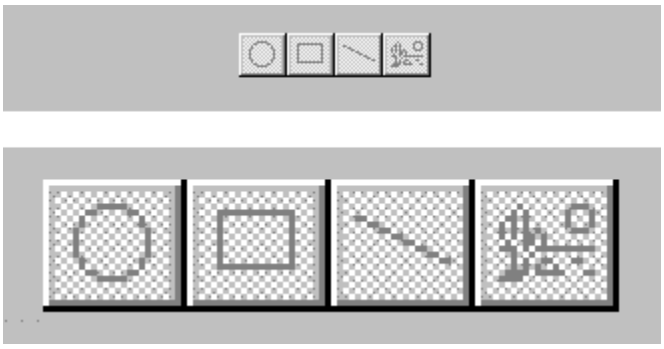


**Figure 13.15 Mixed-value appearance for buttons**

For check box and command button controls displaying mixed-value appearance, when the user clicks the button, the property value or state is set. Clicking a second time clears the value. As an option, you can support a third click to return the button to the mixed-value state.

## Unavailable Appearance

When a control is unavailable (also referred to as disabled), its normal functionality is no longer available to the user (though it can still support access to contextual Help information) because the functionality represented does not apply or is inappropriate under the current circumstances. To reflect this state, the label of the control is rendered with a special *unavailable appearance*, as shown in Figure 13.16.


**Figure 13.16 Unavailable appearance for check boxes and option buttons**

For graphical or text buttons, create the engraved effect by converting the label to monochrome and drawing it in the button highlight color. Then overlay it, at a small offset, with the label drawn in the button shadow color, as shown in Figure 13.17.

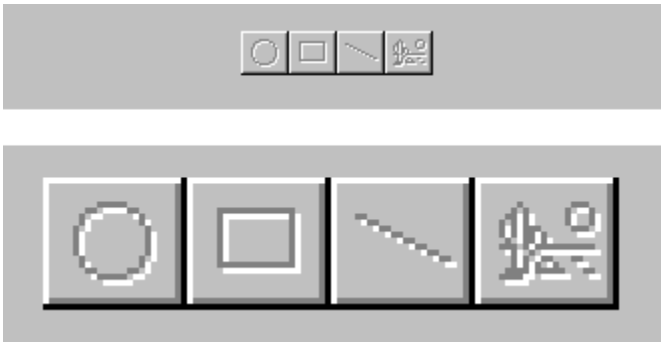


**Figure 13.17 Unavailable appearance for buttons**

If a check box or option button is set, but the control is unavailable, then the control's label is displayed with an unavailable appearance, and its mark appears in the button shadow color, as shown in Figure 13.18.


**Figure 13.18 Unavailable appearance for check boxes and option buttons (when set)**

If a graphical button needs to reflect both the set and unavailable appearance (as shown in Figure 13.19), omit the background checkerboard pattern and combine the option-set and the unavailable appearance for the button's label.

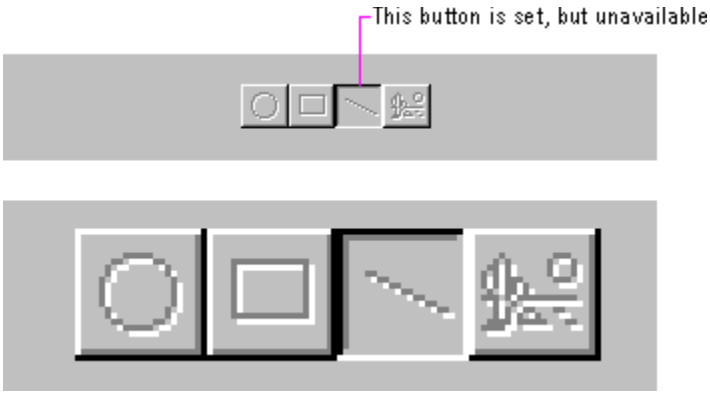



**Figure 13.19 Unavailable and option-set appearance for buttons**

## Input Focus Appearance

You can provide a visual indication so the user knows where the input focus is. For text boxes, the system provides a blinking cursor, or insertion point. For other controls a dotted outline is drawn around the control or the control's label, as shown in Figure 13.20.

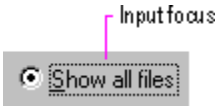



**Figure 13.20 Example of input focus in a control**

The system provides the input focus appearance for standard controls. To use it with your own custom controls, specify the rectangle to allow at least one pixel of space around the extent of the control. If the input focus indicator would be too intrusive, as an option, you can include it around the label for the control. Display the input focus when the mouse button (pen tip) is pressed while over a control; for the keyboard, display the input focus when a navigation or access key for the control is pressed.

The system provides support for drawing the dotted outline input focus indicator using the **DrawFocusRect** function. For more information about this function, see the documentation included in the Win32 SDK.

## Flat Appearance

When you nest controls inside of a scrollable region or control, avoid using a three-dimensional appearance because it may not work effectively against the background. Instead, use the flat appearance style, as shown in Figure 13.21.

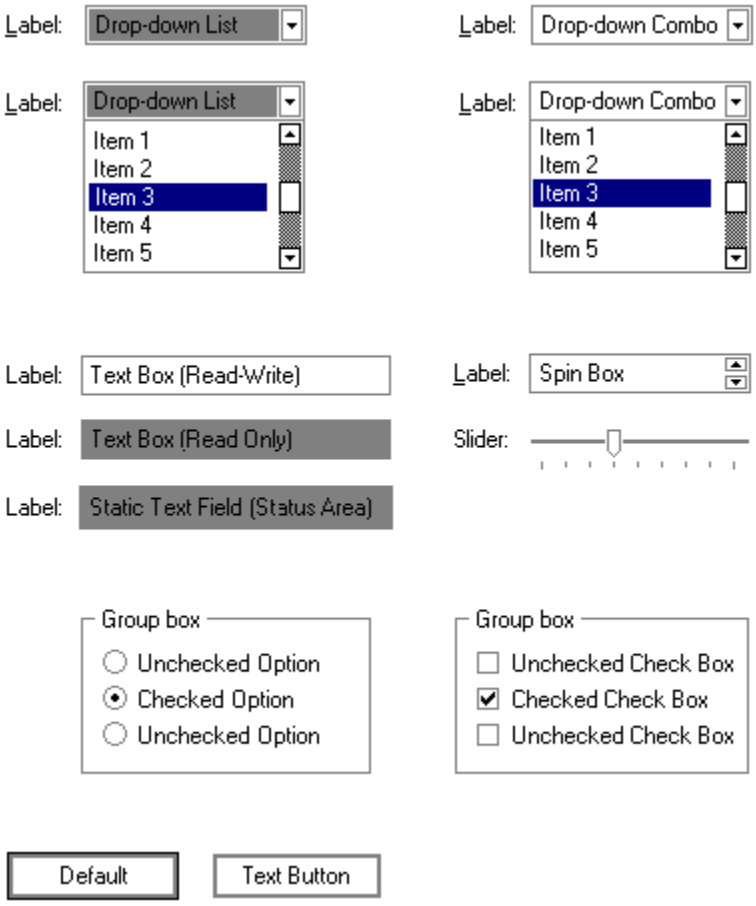



**Figure 13.21 Flat appearance for standard controls**

The system provides support for the flat appearance style for standard controls. It also includes support for drawing the edges of your own custom controls so you can match the appearance used by standard system controls.

The **DrawFrameControl** and **DrawEdge** functions support drawing the flat appearance. For more information about these functions, see the documentation included in the Win32 SDK.

## Layout

Size, spacing, and placement of information are critical in creating a visually consistent and predictable environment. Visual structure is also important for communicating the purpose of the elements displayed in a window. In general, follow the layout conventions for how information is read. In western countries, this means left-to-right, top-to-bottom, with the most important information located in the upper left corner.

## Font and Size

The default system font is a key metric in the presentation of visual information. The default font used for interface elements in Windows (U.S. release) is MS® Sans Serif for 8-point. Menu bar titles, menu items, control labels, and other interface text all use 8-point MS Sans Serif. Title bar text also uses the 8-point MS Sans Serif bold font, as shown in Figure 13.22. However, because the user can change the system font, make certain you check this setting and adjust the presentation of your interface appropriately rather than assuming a fixed size for fonts or other visual elements. Also adjust your presentation when the system notifies your application that these settings have changed.

The **GetSystemMetrics** (standard windows elements), **SystemParametersInfo** (primary windows fonts), and **GetStockObject** (secondary windows fonts) functions provide the current system settings. The WM_SETTINGCHANGE message notifies applications when these settings change. For more information about these APIs, see the documentation included in the Win32 SDK.

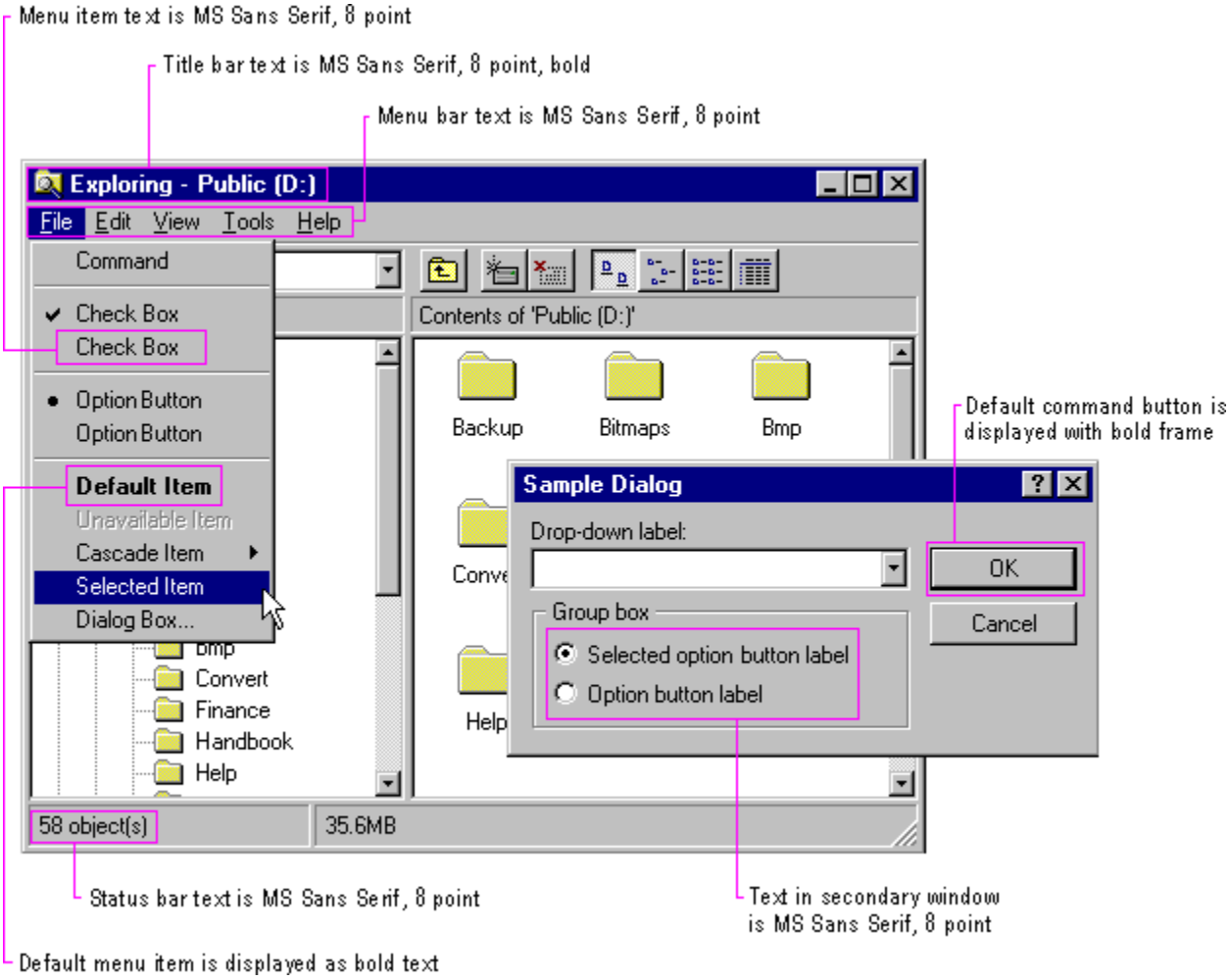



**Figure 13.22 Default font usage in windows**

The system also provides settings for the font and size of many system components including title bar height, menu bar height, border width, title bar button height, icon title text, and scroll bar height and width. When designing your window layouts, take these variables into consideration so that your interface will scale appropriately. In addition, use these standard system settings to determine the size

of your custom interface elements.

The system defines the size and location of user-interface elements in a window based on dialog base units, not pixels. A *dialog base unit* is the device-independent measure to use for layout. One horizontal dialog base unit is equal to one-fourth of the average character width for the current system font. One vertical dialog base unit is equal to one-eighth of an average character height for the current system font. The default height for most single-line controls is 14 dialog base units. Be careful when using a pixel-based drawing program because this may not provide an accurate representation when you translate your design into dialog base units.

Your application can retrieve the number of pixels per base unit for the current display using the **GetDialogBaseUnits** function. For more information about this function, see the documentation included in the Win32 SDK.

If a menu item represents a default command, the text is bold. Default command buttons use a bold outline around the button. In general, use nonbold text in your windows. Use bold text only when you want to call attention to an area or create a visual hierarchy.

Avoid displaying any secondary window larger than 263-dialog bases units x 263-dialog base units. The recommended sizes for property sheets are 252-dialog base units wide x 218-dialog base units high, 227-dialog bases units wide x 215-dialog base units high, and 212-dialog base units x 188-dialog base units high. These sizes keep the window from becoming too large to display at some resolutions, and still provide reasonable space to display supportive information, such as Help windows, that apply to the dialog box or property sheet.

For easy readability, make buttons a consistent length. However, if maintaining this consistency greatly expands the space required by a set of buttons, it may be reasonable to have one button larger than the rest.

Similarly, when using tabs, try to maintain a consistent width for all tabs in the same window (and same dimension). However, if a particular tab's label makes this unworkable, you can size it larger, and maintaining a smaller, consistent size for the other tabs. If a tab's label contains variable text, you may want to size the tab to fit the label, up to some reasonable maximum, after which you truncate and add an ellipsis.

Provide toolbar buttons in at least two different sizes: 24-pixels wide x 22-pixels high and 32-pixels wide x 30-pixels high. This includes the border. The image size you include as the button's label should be 16 x 16 pixels and 24 x 24 pixels, respectively. It is best to center the image on the button's face. Use the smaller size as your default presentation and provide an option for users to change the size. Be certain that you include button images to support all visual states for the buttons.

For more information about common toolbar images, see Menus, Controls, and Toolbars.

For larger buttons on very high resolution displays, you can proportionally size the button to be the same height as a text box control. This allows the button to maintain its proportion with respect to other controls in the toolbar. You can stretch the image when the button is an even multiple of the basic sizes. Alternatively, you can supply additional image sizes. This may be preferable, because it provides better visual results.

Toolbar buttons generally have only graphical labels and no accompanying textual label. You can use a tooltip to provide the name of the button.

## Capitalization

When displaying text in menus, command buttons, and tabs, use conventional book title capitalization. For example, for U.S. versions, capitalize the first letter in each word unless it is an article or preposition not occurring at the beginning or end of the name, or unless the word's conventional usage is not capitalized. For example:

Insert Object
Paste Link
Save As
Go To
Always on Top
By Name

Use this same convention for default names you provide for filenames, title bar text, or icon labels. Of course, if the user supplies a name for an object, display the name as the user specifies it, regardless of case.

Field labels, such as those used for option buttons, check boxes, text boxes, group boxes, and page tabs, should use sentence-style capitalization. For U.S. versions, this means capitalize only the first letter of the initial word and any words that are normally capitalized. For example:

Extended (XMS) memory
Working directory
Print to
Find whole words only

## Grouping and Spacing

Group related components together. You can use group box controls or spacing. Leave at least four dialog base units between controls. Although you can also use color to visually group objects, it is not a common convention and could result in undesirable effects when the user changes color schemes.

Maintain a consistent margin (seven dialog base units is recommended) from the edge of the window. Use spacing between groups within the window. Figure 13.23 shows the recommended spacing.

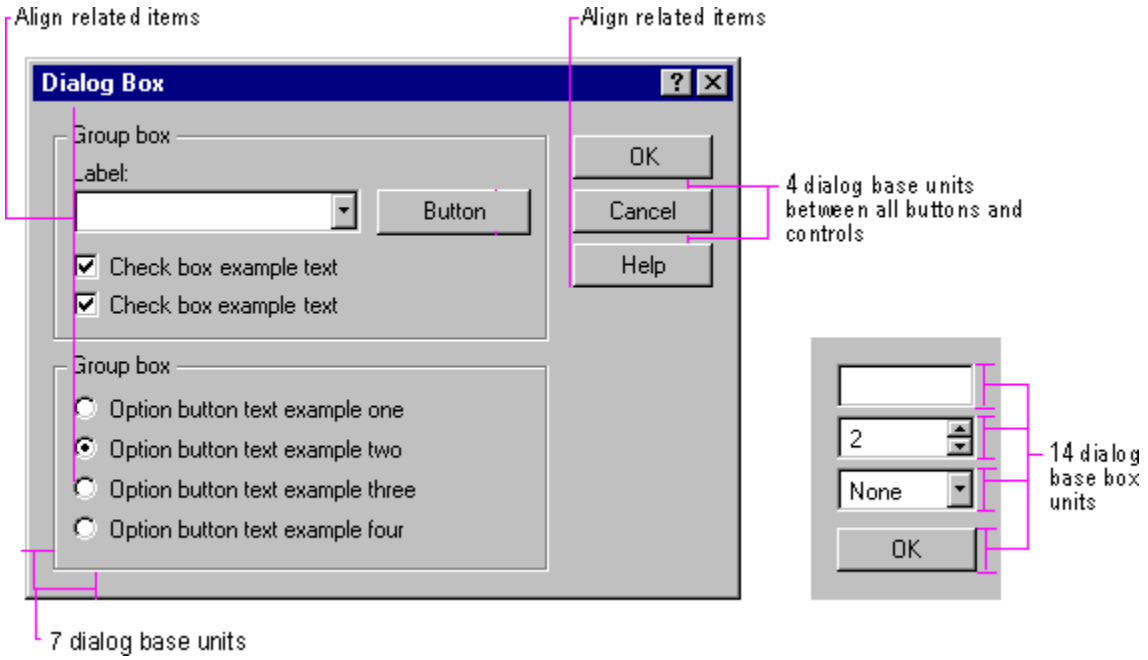



**Figure 13.23 Recommended layout and spacing of controls and text**

Position controls in a toolbar so that there is at least a window's border width from the edges of the toolbar. Use at least four dialog base units spacing between controls, unless you want to align a set of related toolbar buttons adjacently. Use adjacent alignment for toolbar buttons that are related. For example, when using toolbar buttons like a set of option buttons, align them without any spacing between them.

## Alignment

When information is positioned vertically, align fields by their left edges (in western countries). This usually makes it easier for the user to scan the information. Text labels are usually left aligned and placed above or to the left of the areas to which they apply. When placing text labels to the left of text box controls, align the height of the text with text displayed in the text box.

## Placement

Stack the main command buttons in a secondary window in the upper right corner or in a row along the bottom, as shown in Figure 13.24. If there is a default button, it is typically the first button in the set. Place OK and Cancel buttons next to each other. The last button is a Help button (if supported). If there is no OK button, but other command buttons, it is best to place the Cancel button at the end of a set of action buttons, but before a Help button. If a particular command button applies only to a particular field, group it with that field.

For more information about button placement in secondary windows, see Secondary Windows.


**Figure 13.24 Examples of layout of buttons**

Placement of command buttons (or other controls) within a tabbed page implies the application of only the transactions on that page. If command buttons are placed within the window, but not on the tabbed page, they apply to the entire window.

# Design of Graphic Images

When designing pictorial representations of objects, whether they are icons or graphical buttons, begin by defining the icon's purpose and its use. Brainstorm about possible ideas, considering real-world metaphors. It is often difficult to design icons that define operations or processes — activities that rely on verbs. Consider nouns instead. For example, scissors can represent the action of Cut.

Draw your ideas using an icon-editing utility or pixel (bitmapped) drawing package. Drawing them directly on the screen provides immediate feedback about their appearance. It is a good idea to begin the design in black and white. Consider color as an enhancing property. Also, test your images on different backgrounds. They may not always be seen against white or gray backgrounds.

Consistency is also important in the design of graphic images. As with other interface elements, design images assuming a light source from the upper left. In addition, make certain the scale (size) and orientation of your graphics are consistent with the other objects to which they are related and fit well within the working environment.

Avoid using a triangular arrow graphic similar to the one used in cascading menus, drop-down controls, and scroll arrows. When this image appears on a button, it implies that the control will display additional information. For example, you can use an arrow graphic to designate a menu button.

You may want to use a technique called anti-aliasing when designing graphic images. *Anti-aliasing* involves adding colored pixels to smooth the jagged edges of a graphic. However, avoid using anti-aliasing on the outside edge of an icon because the contrasting pixels may look jagged or fuzzy on varying backgrounds.

Finally, remember to consider the potential cultural impact of your graphics. What may have a certain meaning in one country or culture may have unforeseen meanings in another. It is best to avoid letters or words, if possible, as this may make the graphics difficult to apply for other cultures.

For more information about designing for international audiences, see Special Design Considerations.

# Icon Design

Icons are used throughout the Windows interface to represent objects or tasks. Because the system uses icons to represent your software's objects, it is important to not only supply effective icons, but to design them to effectively communicate their purpose.

When designing icons, design them as a set, considering their relationship to each other and to the user's tasks. Do several sketches or designs and test them for usability.

## Sizes and Types

Supply icons for your application in all standard sizes: 16 x 16 pixel (16 color), 32 x 32 pixel (16 color), and 48 x 48 pixel (256 color), as shown in Figure 13.25. Although you can also include greater color depth for the smaller sizes, it increases the storage requirements for the icons and may not be displayable on all computer configurations. Therefore, if you choose to provide 256 color icons in the smaller sizes, do so in addition to providing the standard (16 color) format.

To display icons at 48-x48-pixel resolution, the registry value Shell Icon Size, must be increased to 48. To display icons in color resolution depth higher than 16 colors, the registry value Shell Icon BPP must be set to 8 or more. These values are stored in **HKEY_ CURRENT_USER\Desktop\WindowMetrics**.

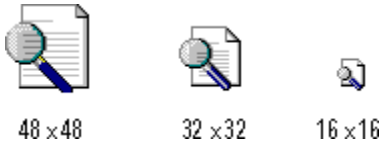



**Figure 13.25 Three sizes of icons**

The system automatically maps colors in your icon design for monochrome configurations. However, you should check your icon design in monochrome configuration. If the result is not satisfactory, author and supply monochrome icons as well.

Define icons not only for your application executable file, but also for all data file types supported by your application, as shown in Figure 13.26.

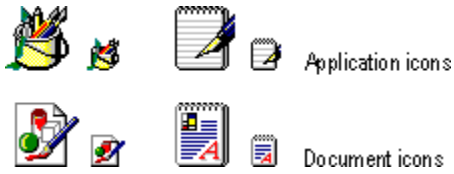



**Figure 13.26 Application and supported document icons**

Icons for documents and data files should be distinct from the application's icon. Include some common element of the application's icon, but focus on making the document icon recognizable and representative of the file's content.

Register the icons you supply in the system registry. If your software does not register any icons, the system automatically provides one based on your application's icon, as shown in Figure 13.27. However, it is unlikely to be as detailed or distinctive as one you can supply.

For more information about registering your icons, see Integrating with the System.

**Figure 13.27 System-generated icon for file type without a registered icon**

## Icon Style

When designing your icons, use a common style across all icons. Repeat common characteristics, but avoid repeating unrelated elements.

Previous to Microsoft Windows 95, black outlines were recommended for icon design. This style recommendation has been dropped.

An illustrative style tends to communicate metaphorical concepts more effectively than abstract symbols. However, in designing an image based on a real-world object, use only the amount of detail that is really necessary for user recognition and recall. Where possible and appropriate, use perspective and dimension (lighting and shadow) to better communicate the real-world representation, as shown in Figure 13.28.

**Figure 13.28 Perspective and dimension improve graphics**

User recognition and recollection are two important factors to consider in icon design. Recognition means that the icon is identifiable by the user and easily associated with a particular object. Support user recognition by using effective metaphors. Use real-world objects to represent abstract ideas so that the user can draw from previous learning and experiences. Exploit the user's knowledge of the world and allude to the familiar.

To facilitate recollection, design your icons to be simple and distinct. Applying the icon consistently also helps build recollection; therefore, design your small icons to be as similar as possible to their larger counterparts. It is generally best to try to preserve general shape and any distinctive detail. 48 x 48-pixel icons can be rendered in 256 colors. This allows very realistic-looking icons, but focus on simplicity and careful use of color. If your software is targeted at computers that can only display 256 colors, make certain you only use colors from the system's standard 256-color palette. If you aim at computers configured for 65,000 or more colors, you can use any combination of colors.

## Pointer Design

You can use a pointer's design to help the user identify objects and provide feedback about certain conditions or states. However, use pointer changes conservatively so that the user is not distracted by excessive flashing of multiple pointer changes while traversing the screen. One way to handle this is to use a time-out before making noncritical pointer changes.

For more information about some of the common pointers, see Input Basics. For information about displaying pointers for drag and drop operations, see General Interaction Techniques.

When you use a pointer to provide feedback, use it only over areas where that state applies. For example, when using the hourglass pointer to indicate that a window is temporarily noninteractive, if the pointer moves over a window that is interactive, change it to its appropriate interactive image. If a process makes the entire interface non-interactive, display the hourglass pointer wherever the user moves the pointer.

Pointer feedback may not always be sufficient. For example, for processes that last longer than a few seconds, it is better to use a progress indicator that indicates progressive status, elapsed time, estimated completion time, or some combination of these to provide more information about the state of the operation. In other situations, you can use command button states to reinforce feedback; for example, when the user chooses a drawing tool.

Use a pointer that best fits the context of the activity. The I-beam pointer is best used to select text. The normal arrow pointer works best for most drag and drop operations, modified when appropriate to indicate copy and link operations.

The location for the hot spot of a pointer (shown in Figure 13.29) is important for helping the user target an object. The pointer's design should make the location of the hot spot intuitive. For example, for a cross-hair pointer, the implied hot spot is the intersection of the lines.

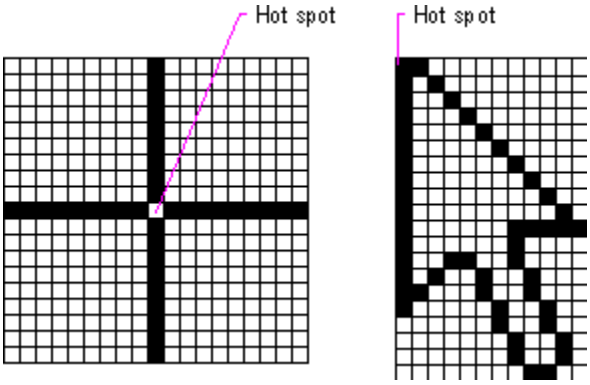



**Figure 13.29 Pointer hot spots**

Animating a pointer can be a very effective way of communicating information. However, remember that the goal is to provide feedback, not to distract the user. In addition, pointer animation should not restrict the user's ability to interact with the interface.

# Selection Appearance

When the user selects an item, provide visual feedback to enable the user to distinguish it from items that are not selected. Selection appearance generally depends on the object and the context in which the selection appears.

Display an object with selection appearance as the user performs a selection operation. For example, display selection appearance when the user presses the mouse button to select an object.

For more information about selection techniques, see Input Basics.

It is best to display the selection appearance only for the scope, area, or level (window or pane) that is active. This helps the user recognize which selection currently applies and the extent of the scope of that selection. Therefore, avoid displaying selections in inactive windows or panes, or at nested levels.

However, in other contexts, it may still be appropriate to display selection appearance simultaneously in multiple contexts. For example, when the user selects an object and then selects a menu item to apply to that object, selection appearance is always displayed for both the object and the menu item because it is clear where the user is directing the input. In cases where you need to show simultaneous selection, but with the secondary selection distinguished from the active selection, you can draw an outline in the selection highlight color around the secondary selection or use some similar variant of the standard selection highlight technique.

## Highlighting

For many types of objects, you can display the object or its background or some distinguishing part of the object using the system highlight color. Figure 13.30 shows examples of selection appearances.

The **GetSysColor** function provides access to the current setting for the system selection highlight color (COLOR_HIGHLIGHT). For more information about this function, see the documentation included in the Win32 SDK.

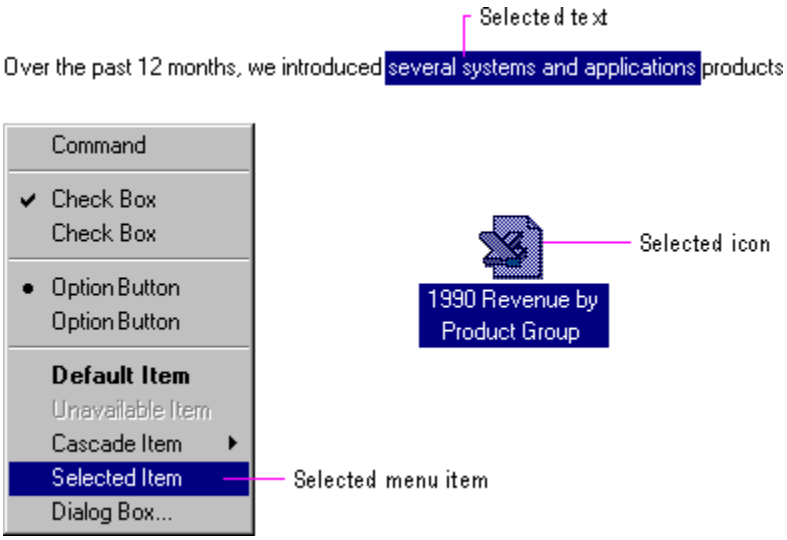



**Figure 13.30 Examples of selection appearance**

In a secondary window, it may be appropriate to display selection highlighting when the highlight is also being used to reflect the setting for a control. For example, in list boxes, highlighting often indicates a current setting. In cases like this, provide an input focus indication as well so the user can distinguish when input is being directed to another control in the window; you can also use check marks instead of highlighting to indicate the setting.

## Handles

Handles provide access to operations for an object, but they can also indicate selection for some kinds of objects. The typical handle is a solid, filled square box that appears on the edge of the object, as shown in Figure 13.31.

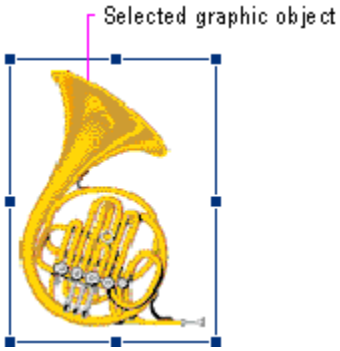



**Figure 13.31 Selected graphic object with handles**

The handle is “hollow” when the handle indicates selection, but is not a control point by which the object may be manipulated. Figure 13.32 shows a solid and a hollow handle.

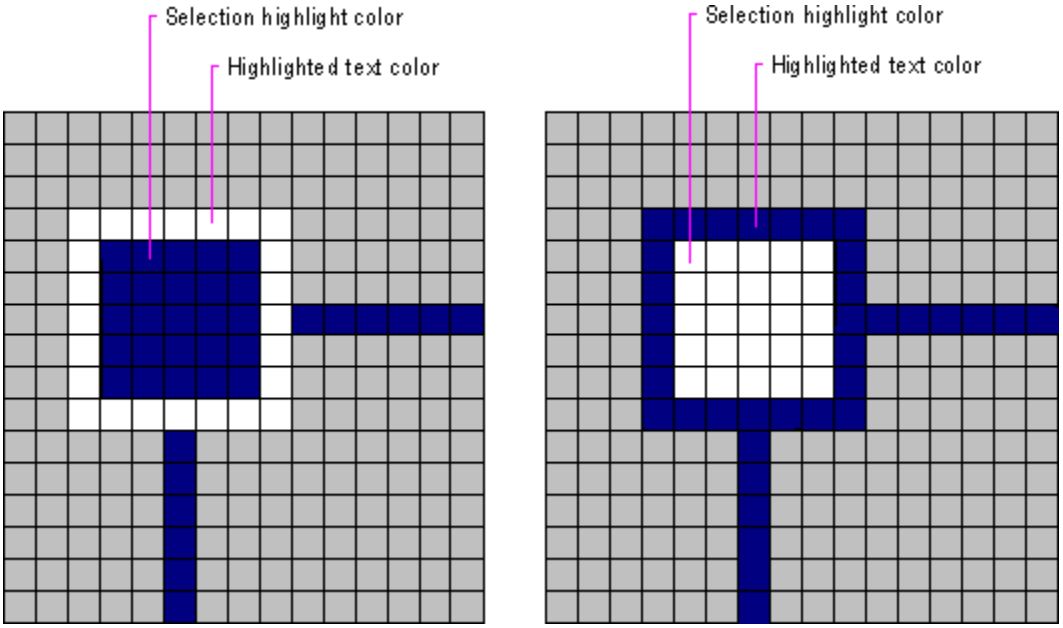



**Figure 13.32 Solid and hollow handles**

Base the default size of a handle on the current system settings for window border and edge metrics so that your handles are appropriately sized when the user explicitly changes window border widths or to accommodate higher resolutions. Similarly, base the colors you use to draw handles on system color metrics so that when the user changes the default system colors, handles change appropriately.

The system settings for window border and edge metrics can be accessed using the **GetSystemMetrics** function. For more information about this function, see the documentation included in the Win32 SDK.

When using handles to indicate selection, display the handle in the system highlight color. To help distinguish the handle from the variable background, draw a border and the edge of the handle using the system's setting for highlighted text. For hollow handles use the opposite: selection highlight color for the border and highlighted text color for the fill color. If you display handles for an object even when it is not selected, display handles in a different color, such as the window text color, so that the user will not confuse it as part of the active selection.

## Transfer Appearance

When the user drags an object to perform an operation, for example, move, copy, print, and so on, display a representation of the object that moves with the pointer. In general, do not simply change the pointer to be the object, as this may obscure the insertion point at some destinations. Instead, use a translucent or outline representation of the object that moves with the pointer, as shown in Figure 13.33.

For more information about drag and drop transfer operations, see General Interaction Techniques.

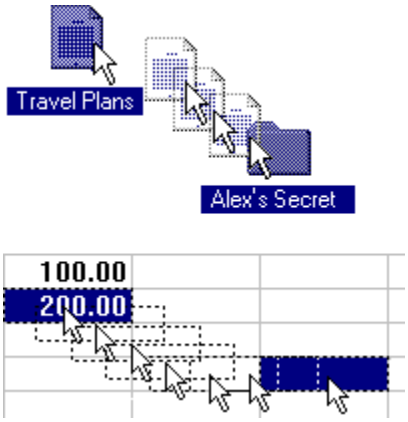



**Figure 13.33 Translucent and outline representation (drag transfer)**

You can create a translucent representation by using a checkerboard mask made up of 50 percent transparent pixels. When used together with the object's normal appearance, this provides a representation that allows part of the destination to show through. An outline representation should also use a translucent or transparent interior and a gray or dotted outline.

The presentation of an object being transferred displayed is always defined by the destination. Use a representation that best communicates how the transferred object will be incorporated when the user completes the drag transfer. For example, if the object being dragged will be displayed as an icon, then display an icon as its representation. If, on the other hand, it will be incorporated as native content, then display an appropriate representation. For example, you could display a graphics object as an outline or translucent image of its shape, a table cell as the outline of a rectangular box, and text selection as the first few characters of a selection with a transparent background.

Set the pointer image to be whatever pointer the target location uses for directly inserting information. For example, when dragging an object into normal text context, use the I-beam pointer. In addition, include the copy or link image at the bottom right of the pointer if that is the interpretation for the operation.

## Open Appearance

Open appearance is most commonly used for an OLE embedded object, but it can also apply in other situations where the user opens an object into its own window. To indicate that an object is “open,” display the object in its container’s window overlaid with a set of hatched (45 degree) lines drawn every four pixels, as shown in Figure 13.34.

For more information about the use of open appearance for OLE embedded objects, see Working with OLE Embedded and OLE Linked Objects.

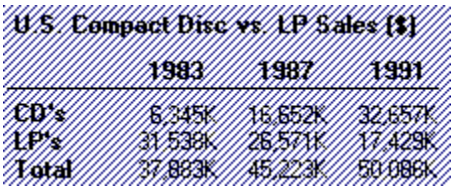



**Figure 13.34 An object with opened appearance**

If the opened object is selected, display the hatched lines using the system highlight color. When the opened object is not selected, display the lines using the system’s setting for window text color.

# Animation

Animation can be an effective way to communicate information. For example, it can illustrate the operation of a particular tool or reflect a particular state. It can also be used to include an element of fun in your interface. You can use animation effects for objects within a window and interface elements, such as icons, buttons, and pointers.

Effective animation involves many of the same design considerations as other graphics elements, particularly with respect to color and sound. Fluid animation requires presenting images at 16 (or more) frames per second.

When you add animation to your software, ensure that it does not affect the interactivity of the interface. Do not force the user to remain in a modal state to allow the completion of the animation. Unless animation is part of a process, make it interruptible by the user or independent of the user’s primary interaction.

Avoid gratuitous use of animation. When animation is used for decorative effect it can distract or annoy the user. You may want to provide the user with the option of turning off the animation or otherwise customizing the animation effects.

# Introduction

A well-designed application for Microsoft Windows must consider other factors to appeal to the widest possible audience. This topic covers special user interface design considerations, such as support for sound, accessibility, internationalization, and network computing.

 Adding Menu Items and Toolbar Buttons

 Supporting Connections

 Installing Information Services

## Sound

You can incorporate sound as a part of an application in several ways — for example, music, speech, or sound effects. Such auditory information can take the following forms:

- A primary form of information, such as the composition of a particular piece of music or a voice message.
- An enhancement of the presentation of information that is not required for the operation of the software.
- A notification or alerting of users to a particular condition.

Sound can be an effective form of information and interface enhancement when appropriately used. However, avoid using sound as the only means of conveying information. Some users may be hard-of-hearing or deaf. Others may work in a noisy environment or in a setting that requires that they disable sound or maintain it at a low volume. In addition, like color, sound is a very subjective part of the interface.

As a result, sound is best incorporated as a redundant or secondary form of information, or supplemented with alternative forms of communication. For example, if a user turns off the sound, consider flashing the window's title bar, taskbar button, presenting a message box, or other means of bringing the user's attention to a particular situation. Even when sound is the primary form of information, you can supplement the audio portion by providing visual representation of the information that might otherwise be presented as audio output, such as captioning or animation.

The taskbar can also provide visual status or notification information. For more information about using the taskbar for this purpose, see Integrating with the System.

Always allow the user to customize sound support. Support the standard system interfaces for controlling volume and associating particular sounds with application-specific sound events. You can also register your own sound events for your application.

The system provides a global system setting, ShowSounds. The setting indicates that the user wants a visual representation of audio information. Your software should query the status of this setting and provide captioning for the output of any speech or sounds. Captioning should provide as much information visually as is provided in the audible format. It is not necessary to caption ornamental sounds that do not convey useful information.

The **GetSystemMetrics** function provides access to the ShowSounds and SoundSentry settings. For more information about this function and the settings, see the documentation included in the Microsoft Win32 Software Development Kit (SDK).

Do not confuse ShowSounds with the system's SoundSentry option. When the user sets the SoundSentry option, the system automatically supplies a visual indication whenever a sound is produced. Avoid relying on SoundSentry alone if the ShowSounds option is set because SoundSentry only provides rudimentary visual indications, such as flashing of the display or screen border, and it cannot convey the meaning of the sound to the user. The system provides SoundSentry primarily for applications that do not provide support for ShowSounds. The user sets either of these options with the Microsoft Windows Accessibility Options.

In Microsoft Windows 95, SoundSentry only works for audio output directed through the internal PC speaker.

# Accessibility

*Accessibility* means making your software usable and accessible to a wide range of users, including those with disabilities. Many users may require special accommodation because of temporary or permanent disabilities.

The issue of software accessibility in the home and workplace is becoming increasingly important. Nearly one in five Americans have some form of disability — and it is estimated that 30 million people in the U.S. alone have disabilities that may be affected by the design of your software. In addition, between seven and nine out of every ten major corporations employ people with disabilities who may need to use computer software as part of their jobs. As the population ages and more people become functionally limited, accessibility for users with disabilities will become increasingly important to the population as a whole. Legislation, such as the Americans with Disabilities Act, requires that most employers provide reasonable accommodation for workers with disabilities. Section 508 of the Rehabilitation Act is also bringing accessibility issues to the forefront in government businesses and organizations receiving government funding.

Designing software that is usable for people with disabilities does not have to be time consuming or expensive. However, it is much easier if you include this in the planning and design process rather than attempting to add it after the completion of the software. Following the principles and guidelines in this guide will help you design software for most users. Often recommendations, such as the conservative use of color or sound, often benefit all users, not just those with disabilities. In addition, keep the following basic objectives in mind:

- Provide a customizable interface to accommodate a wide variety of user needs and preferences.
- Provide compatibility with accessibility utilities that users install.
- Avoid creating unnecessary barriers that make your software difficult or inaccessible to certain types of users.

The following sections provide information on types of disabilities and additional recommendations about how to address the needs of customers with those disabilities.

## Types of Disabilities

There are many types of disabilities, but they are often grouped into several broad categories. These include visual, hearing, physical movement, speech or language impairments, and cognitive and seizure disorders.

## Visual Disabilities

Visual disabilities range from slightly reduced visual acuity to total blindness. Those with reduced visual acuity may only require that your software support larger text and graphics. For example, the system provides scalable fonts and controls to increase the size of text and graphics. To accommodate users who are blind or have severe impairments, make your software compatible with speech or Braille utilities, described in a later topic.

Color blindness and other visual impairments may make it difficult for users to distinguish between certain color combinations. This is one reason why color is not recommended as the only means of conveying information. Always use color as an additive or enhancing property.

## Hearing Disabilities

Users who are deaf or hard-of-hearing are generally unable to detect or interpret auditory output at normal or maximum volume levels. Avoiding the use of auditory output as the only means of communicating information is the best way to support users with this disability. Instead, use audio output only as a redundant, additive property or provide visual output as an option to supplement the audio information. For more information about supporting sound, see Sound.

## Physical Movement Disabilities

Some users have difficulty or are unable to perform certain physical tasks — for example, moving a mouse or simultaneously pressing two keys on the keyboard. Other individuals have a tendency to inadvertently strike multiple keys when targeting a single key. Consideration of physical ability is important not only for users with disabilities, but also for beginning users who need time to master all the motor skills necessary to interact with the interface. The best way to support these users is by supporting all your basic operations using simple keyboard and mouse interfaces.

## Speech or Language Disabilities

Users with language disabilities, such as dyslexia, find it difficult to read or write. Spell- or grammar-check utilities can help children, users with writing impairments, and users with a different first language. Supporting accessibility tools and utilities designed for users who are blind can also help those with reading impairments. Most design issues affecting users with oral communication difficulties apply only to utilities specifically designed for speech input.

## Cognitive Disabilities

Cognitive disabilities can take many forms, including perceptual differences and memory impairments. You can accommodate users with these disabilities by allowing them to modify or simplify your software's interface, such as supporting menu or dialog box customization. Similarly, using icons and graphics to illustrate objects and choices can be helpful for users with some types of cognitive impairments.

## Seizure Disorders

Some users are sensitive to visual information that alternates its appearance or flashes at particular rates — often the greater the frequency, the greater the problem. However, there is no perfect flash rate. Therefore, base all modulating interfaces on the system's cursor blink rate. Because users can customize this value, a particular frequency can be avoided. If that is not practical, provide your own interface for changing the flash rate.

The **GetCaretBlinkTime** function provides access to the current cursor blink rate setting. For more information about this function, see the documentation included in the Win32 SDK.

## Types of Accessibility Aids

There are a number of accessibility aids to assist users with certain types of disabilities. To allow these users to effectively interact with your application, make certain it is compatible with these utilities. This section briefly describes the types of utilities and how they work.

One of the best ways to accommodate accessibility in your software's interface is to use standard Windows conventions wherever possible. Windows already provides a certain degree of customization for users and most accessibility aids work best with software that follows standard system conventions.

## Screen Enlargement Utilities

Screen enlargers (also referred to as screen magnification utilities or large print programs) allow users to enlarge a portion of their screen. They effectively turn the computer monitor into a viewport showing only a portion of an enlarged virtual display. Users then use the mouse or keyboard to move this viewport to view different areas of the virtual display. Enlargers also attempt to track where users are working, following the input focus and the activation of windows, menus, and secondary windows, and can automatically move the viewport to the active area.

## Screen Review Utilities

People who cannot use the visual information on the screen can interpret the information with the aid of a screen review utility (also referred to as a screen reader program or speech access utility). Screen review utilities take the displayed information on the screen and direct it through alternative media, such as synthesized speech or a refreshable Braille display. Because both of these media present only text information, the screen review utility must render other information on the screen as text; that is, determine the appropriate text labels or descriptions for graphical screen elements. They must also track users' activities to provide descriptive information about what the user is doing. These utilities often work by monitoring the system interfaces that support drawing on the screen. They build an off-screen database of the objects on the screen, their properties, and their spatial relationships. Some of this information is presented to

users as the screen changes, and other information is maintained until users request it. Screen review utilities often include support for configuration files (also referred to as set files or profiles) for particular applications.

## Voice Input Systems

Users who have difficulty typing can choose a voice input system (also referred to as a speech recognition program) to control software with their voice instead of a mouse and keyboard. Like screen reader utilities, voice input systems identify objects on the screen that users can manipulate. Users activate an object by speaking the label that identifies the object. Many of these utilities simulate keyboard interfaces, so if your software includes a keyboard interface, it can be adapted to take advantage of this form of input.

## On-Screen Keyboards

Some individuals with physical disabilities cannot use a standard keyboard, but can use special devices designed to work with an on-screen keyboard. Switching devices display groups of commands displayed on the screen, and the user employs one or more switches to choose a selected group, then a command within the group. Another technique allows a user to use a special mouse or headpointer (a device that lets users manipulate the mouse pointer on the screen through head motion) to point to graphic images of keys displayed on the screen to generate keystroke input.

## Keyboard Filters

Impaired physical abilities, such as erratic motion, tremors, or slow response, can sometimes be compensated by filtering out inappropriate keystrokes. The Windows Accessibility Options supports a wide range of keyboard filtering options. These are generally independent of the application with which users are interacting and therefore require no explicit support except for the standard system interfaces for keyboard input. However, users relying on these features may type slowly.

## Compatibility with Screen Review Utilities

You can use the following techniques to ensure software compatibility with screen review utilities. The system allows your application to determine whether the system has been configured to provide support for a screen review utility, allowing your software to enable or disable certain capabilities.

You can check the SM_SCREENREADER setting using the **GetSystemMetrics** function. For more information about this function and other information about supporting screen review utilities, see the documentation included in the Win32 SDK.

## Controls

Use standard Windows controls wherever possible. Most of these have already been implemented to support screen review and voice input utilities. However, custom controls you create may not be usable by screen review utilities.

Always include a label for every control, even if you do not want the control's label to be visible. This applies regardless of whether you use standard controls or your own specialized controls, such as owner drawn controls or custom controls. If the control does not provide a label, you can create a label using a static text control.

Follow the normal layout conventions by placing the static text label before the control (above or to the left of the control). Also, set the keyboard TAB navigation order appropriately so that tabbing to a label navigates to the associated control it identifies instead of a label. To make certain that the label is recognized correctly, include a colon at the end of the label's text string, unless you are labeling a button, tab, or group box control. In cases where a label is not needed or would be visually distracting, provide the label but do not make it visible. Although the label is not visible, it is accessible to a screen review utility.

Text labels are also effective for choices within a control. For example, you can enhance menus or lists that display colors or line widths by including some form of text representation, as shown in Figure 14.1.

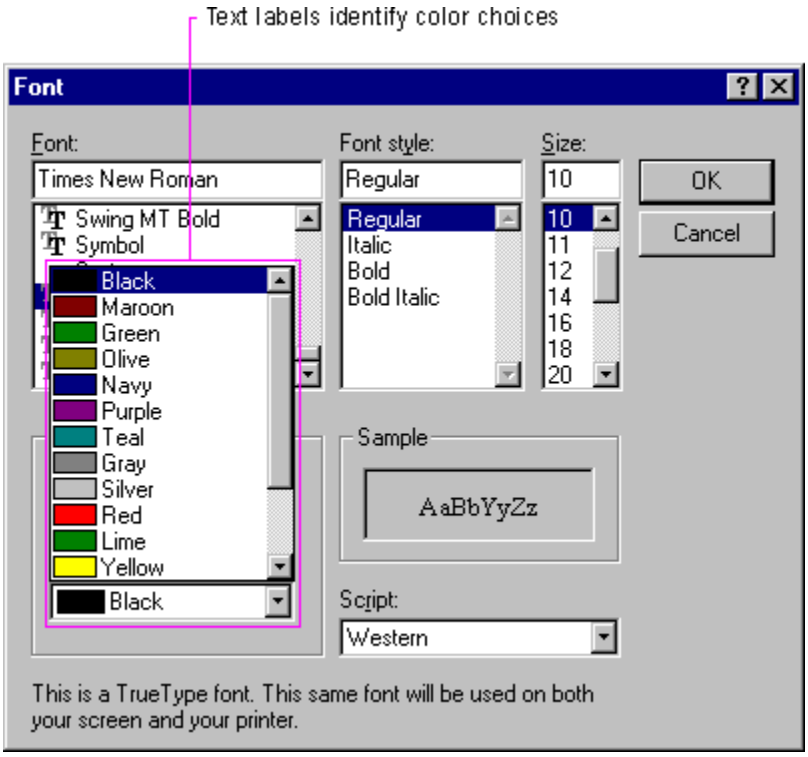



**Figure 14.1 Using text to help identify choices**

If providing a combined presentation is too difficult, offer users the choice between text and graphical representation, or choose one of them based on the system's screen review utility setting.

## Text Output

Screen review utilities usually interpret text — including properties such as font, size, and face — that is displayed with standard system interfaces. However, text displayed as graphics (for example, bitmapped text) is not accessible to a screen review utility. To make it accessible, your application can create an invisible text label and associate the graphical representation of text with it by drawing the text over the graphic with a null operator (NOP). Screen review utilities can read standard text representations in a metafile, so you can also use metafiles, instead of bitmap images, for graphics information that includes text.

## Graphics Output

Users with normal sight may be able to easily distinguish different elements of a graphic or pictorial information, such as a map or chart, even if they are drawn as a single image; however, a screen review utility must distinguish between different components. There are a number of ways to do this. Any of these methods can be omitted when the system's screen review setting is not set.

When using bitmap images, consider separately drawing each component that requires identification. If performance is an issue, combine the component images in an off-screen bitmap using separate drawing operations and then display the bitmap on the screen with a single operation. You can also draw multiple bitmap images with a single metafile.

Alternatively, you can redraw each component separately, or draw a separate image to identify each region using a null operator. This will not have an effect on the visible image, but allows a screen review utility to identify the region. You can also use this method to associate a text label with a graphic element.

When drawing graphics, use standard Windows drawing functions wherever possible. If you change an image directly — for example, clearing a bitmap by writing directly into its memory — a screen review utility will not be able to recognize the content change and will inappropriately describe it to users.

## Icons and Windows

Accompany icons that represent objects with a text label (title) of the object's name. Use the system font and color for icon labels, and follow the system conventions for placement of the text relative to the icon. This allows a screen review utility to identify the object without special support.

Similarly, make certain that all your windows have titles. Even if the title is not visible, it is still available to access utilities. The more unique your window titles, the easier users can differentiate between them, especially when using a screen review utility. Using unique window class names is another way to provide for distinct window identification, but providing appropriate window titles is preferred.

## The User's Point of Focus

Many accessibility aids must follow where the user is working. For example, a screen review utility conveys to users where the input focus is; a screen enlarger pans its viewport to ensure that users' focus is always kept on the visible portion of the screen. Most utilities give users the ability to manually move the viewport, but this becomes a laborious process, especially if it has to be repeated each time the input focus moves.

When the system handles the move of the input focus, such as when the user selects a menu, navigates between controls in a dialog box, or activates a window, an accessibility utility can track the change. However, the utility may not detect when an application moves the input focus within its own window. Therefore, whenever possible, use standard system functions to place the input focus, such as the text insertion point. Even when you provide your own implementation of focus, you can use the system functions to indicate focus location without making the standard input focus indicator visible.

The **SetCaretPos** function is an example of a system function you can use to indicate focus location. For more information about this function, see the documentation included in the Win32 SDK.

## Timing and Navigational Interfaces

Some users read text or press keys very slowly and do not respond to events as quickly as the average user. Avoid displaying critical feedback or messages briefly and then automatically removing them because many users cannot read or respond to them. Similarly, limit your use of time-out based interfaces. If you do include a time-based interface, always provide a way for users to configure the time-out.

Also, avoid displaying or hiding information based on the movement of the pointer, unless it is part of a standard system interface (for example, tooltips). Although such techniques can benefit some users, they may not be available for those using accessibility utilities. If you do provide such support, consider making these features optional so that users can turn them on or off when a screen review utility is installed.

Similarly, you should avoid using general navigation to trigger operations, because users of accessibility aids may need to navigate through all controls. For example, basic TAB keyboard navigation in a dialog box should not activate actions associated with a control, such as setting a check box or carrying out a command button. However, navigation can be used to facilitate further user interaction, such as validating user input or opening a drop-down control.

## Color

Base the color properties of your interface elements on the system colors for window components, rather than defining specific colors. Remember to use appropriate foreground and background color combinations. If the foreground of an element is rendered with the button text color, use the button face color as its background rather than the window background color. If the system does not provide standard color settings that can be applied to some elements, you can include your own interface that allows users to customize colors. In addition, you can provide graphical patterns as an optional substitute for colors as a way to distinguish information.

For more information about the use of color and how it is used for interface elements, see Visual Design.

The system also provides a global setting called High Contrast Mode that users can set through the Windows Accessibility Options. The setting provides contrasting color settings for foreground and background visual elements. Your application should check for this setting's status when it starts, and whenever it receives notification of system setting changes. When set, adjust your interface colors based on those set for the high contrast color scheme. In addition, whenever High Contrast Mode is set, hide any images that are drawn behind text (for example, watermarks or logos) to maintain the legibility of the information on the screen. You can also display monochrome versions of bitmaps and icons using the appropriate foreground color.

The **GetSystemMetrics** function provides access to the SM_HIGHCONTRAST setting. For more information about this function, see the documentation included in the Win32 SDK.

## Scalability

Another important way to provide for visual accessibility is to allow for the scalability of screen elements. Sometimes, this simply means allowing users to change the font for the display of information. The system allows users to change the size and font of standard Windows components. You should use these same metrics for appropriately adjusting the size of other visual information you provide. For your own custom elements, you can provide scaling by including a TrueType font or metafiles for your graphics images.

For more information about the system metrics for font and size, see Visual Design.

It may also be useful to provide scaling features within your application. For example, many application provide a “Zoom” command that scales the presentation of the information displayed in a window, or other commands that make the presentation of information easier to read. You may need to add scroll bars if the scaled information exceeds the current size of the window.

## Keyboard and Mouse Interface

Providing a good keyboard interface is an important step in accessibility because it affects users with a wide range of disabilities. For example, a keyboard interface may be the only option for users who are blind or use voice input utilities, and those who cannot use a mouse. The Windows Accessibility Options often compensate for users with disabilities related to keyboard interaction; however, it is more difficult to compensate for problems related to pointing device input.

You should follow the conventions for keyboard navigation techniques presented in this guide. For specialized interfaces within your software, model your keyboard interface on conventions that are familiar and appropriate for that context. Where they apply, use the standard control conventions as a guide for your defining interaction. For example, support TAB and SHIFT+TAB key and access keys to support navigation to controls.

Make certain the user can navigate to all objects. Avoid relying only on navigational design that requires the user to understand the spatial relationship between objects. Accessibility utilities may not be able to convey such relationships.

Providing a well-designed mouse interface is also important. Pointing devices may be more efficient than keyboards for some users. When designing the interface for pointing input, avoid making basic functions available only through multiple clicking, drag and drop manipulation, and keyboard-modified mouse actions. Such actions are best considered shortcut techniques for more advanced users. Make basic functions available through single click techniques.

The system also allows your application to determine when the user relies on the keyboard, rather than pointing device input. You can use this to present special keyboard interfaces that might otherwise be hidden.

Check the SM_KEYBOARDPREF setting, using **GetSystemMetrics,** to determine whether a user relies on keyboard rather than pointing device input. For more information about this function, see the documentation included in the Win32 SDK.

Where possible, avoid making the implementation of basic functions dependent on a particular device. This is critical for supporting users with physical disabilities and users who may not wish to use or install a particular device.

## Documentation, Packaging, and Support

Although this guide focuses primarily on the design of the user interface, a design that provides for accessibility needs to take into consideration other aspects of a product. For example, consider the documentation needs of your users. For users who have difficulty reading or handling printed material, provide online documentation for your product. If the documentation or installation instructions are not available online, you can provide documentation separately in alternative formats, such as ASCII text, large print, Braille, or audio tape format. Organizations that can help you produce and distribute such documentation are listed in the accessibility section of the Bibliography of this guide.

When possible, choose a format and binding for your documentation that makes it accessible for users with disabilities. As in the interface, information in color should be a redundant form of communication. Bindings that allow a book to lie flat are usually better for users with limited dexterity.

Packaging is also important because many users with limited dexterity can have difficulty opening packages. Consider including an easy-opening overlap or tab that helps users remove shrink-wrapping.

Finally, although support is important for all users, it is difficult for users with hearing impairments to use standard support lines. Consider making these services available to customers using text telephones (also referred to as “TT” or “TDD”). You can also provide support through public bulletin boards or other networking services.

## Usability Testing

Just as it is important to test the general usability of your software, it is a good idea to test how well it provides for accessibility. There are a variety of ways of doing this. One way is to include users with disabilities in your prerelease or usability test activities. In addition, you can establish a working relationship with companies that provide accessibility aids. Information about accessibility vendors or potential test sites is included in the Bibliography.

You can also try running your software in a fashion similar to that used by a person with disabilities. Try some of the following ideas for testing:

- Use the Windows Accessibility Options and set your display to a high contrast scheme, such as white text on a black background. Are there any portions of your software that become invisible or hard to use or recognize?
- Try using your software for a week without using a mouse. Are there operations that you cannot perform? Was anything especially awkward?
- Increase the size of the default system fonts. Does your software still look good? Does your software fonts appropriately adjust to match the new system font?

# Internationalization

To successfully compete in international markets, your software must easily accommodate differences in language, culture, and hardware. This section does not cover every aspect of preparing software for the international market, but it does summarize some of the key design issues.

For more information about the technical details for localizing your application, see the documentation included in the Win32 SDK.

The process of translating and adapting a software product for use in a different country is called *localization*. Like any part of the interface, include international considerations early in the design and development process. In addition to adapting screen information and documentation for international use, Help files, scenarios, templates, models, and sample files should all be a part of your localization planning.

Language is not the only relevant factor when localizing an interface. Several countries can share a common language but have different conventions for expressing information. In addition, some countries can share a language but use a different keyboard convention.

A more subtle factor to consider when preparing software for international markets is cultural differences. For example, users in the U.S. may recognize a rounded mail box with a flag on the side as an icon for a mail program, but this image may not be recognized by users in other countries. Sounds and their associated meanings may also vary from country to country.

It is helpful to create a supplemental document for your localization team that covers the terms and other translatable elements your software uses, and describes where they occur. Documenting changes between versions saves time in preparing new releases.

## Text

A major aspect of localizing an interface involves translating the text used by the software in its title bars, menus and other controls, messages, and some registry entries. To make localization easier, store interface text as resources in your application's resource file rather than including it in the source code of the application. Remember to also translate menu commands your application stores for its file types in the system registry.

Translation is a challenging task. Each foreign language has its own syntax and grammar. Following are some general guidelines to keep in mind for translation:

- Do not assume that a word always appears at the same location in a sentence, that word order is always the same, that sentences or words always have the same length, or that nouns, adjectives, and verbs always keep the same form.
- Avoid using vague words that can have several meanings in different contexts.
- Avoid colloquialisms, jargon, acronyms, and abbreviations.
- Use good grammar. Translation is a difficult enough task without a translator having to deal with poor grammar.
- Avoid dynamic, or run-time, concatenation of different strings to form new strings — for example, composing messages by combining frequently used strings. An exception is the construction of filenames and names of paths.
- Avoid hard coding filenames in a binary file. Filenames may need to be translated.
- Avoid including text in images and icons. Doing so requires that these also be translated.

Translation of interface text from English to other languages often increases the length of text by 30 percent or more. In some extreme cases, the character count can increase by more than 100 percent; for example, the word "move" becomes "verschieben" in German. Accordingly, if the amount of the space for displaying text is strictly limited, as in a status bar, restrict the length of the English interface text to approximately one half of the available space. In contexts that allow more flexibility, such as dialog boxes and property sheets, allow 30 percent for text expansion in the interface design. Message text in message boxes, however, should allow for text expansion of about 100 percent. Avoid having your software rely on the position of text in a control or window because translation may require movement of the text.

Expansion due to translation affects other aspects of your product. A localized version is likely to affect file sizes, potentially changing the layout of your installation disks and setup software.

Translation is not always a one-to-one correspondence. A single word in English can have multiple translations in another language. Adjectives and articles sometimes change spelling according to the gender of the nouns they modify. Therefore, be careful when reusing a string in multiple places. Similarly, several English words may have only a single meaning in another language. This is particularly important when creating keywords for the Help index for your software.

## Graphics

It is best to review the proposed graphics for international applicability early in your design cycle. Localizing graphics can be a time consuming process.

Although graphics communicate more universally than text, graphical aspects of your software — especially icons and toolbar button images — may also need to be revised to address an international audience. For example, a toolbar image that includes a magic wand to represent access to a wizard interface does not have meaning in many European countries and requires a different image.

When possible, choose generic images and glyphs. Even if you can create custom designs for each language, having different images for different languages can confuse users who work with more than one language version.

Many symbols with a strong meaning in one culture do not have any meaning in another. For example, many symbols for U.S. holidays and seasons are not shared around the world. Importantly, some symbols can be offensive in some cultures (for example, the open palm commonly used at U.S. crosswalk signals is offensive in some countries). Some metaphors also may not apply in all languages.

## Keyboards

International keyboards also differ from those used in the U.S. Avoid using punctuation character keys as shortcut keys because they are not always found on international keyboards or easily produced by the user. Remember too, that what seems like an effective shortcut because of its mnemonic association (for example, CTRL+B for Bold) can warrant a change to fit a particular language. Similarly, macros or other utilities that invoke menus or commands based on access keys are not likely to work in an international version, because the command names on which the access keys are based differ.

Keys do not always occupy the same positions on all international keyboards. Even when they do, the interpretation of the unmodified keystroke can be different. For example, on U.S. keyboards, SHIFT+8 results in an asterisk character. However, on French keyboards, it generates the number 8. Similarly, avoid using CTRL+ALT combinations, because the system interprets this combination for some language versions as the ALTGR key, which generates some alphanumeric characters. Similarly, avoid using the ALT key as a modifier because it is the primary keyboard interface for accessing menus and controls. In addition, the system uses many specialized versions for special input. For example, ALT+~ invokes special input editors in Far East versions of Windows. For text fields, pressing ALT+*number* enters characters in the upper range of a character set. Similarly, avoid using the following characters when assigning the following shortcut keys.

@ £ $ { } [ ] \ ~ | ^ ‘ < >

## Character Sets

Some international countries require support for different character sets (sometimes referred to as *code pages*). The system provides a standard interface for supporting multiple character sets and sort tables. Use these interfaces wherever possible for sorting and case conversion. In addition, consider the following guidelines:

For more information about the technical details for localizing your application, see the documentation included in the Win32 SDK.

- Do not assume that the character set is U.S. ANSI. Many ANSI character sets are available. For example, the Russian version of Windows 95 uses the Cyrillic ANSI character set which is different than the U.S. ANSI set.
- Use the system functions for supporting font selection (such as the common font dialog box).
- Always save the character set with font names in documents.

## Formats

Different countries often use substantially different formats for dates, time, money, measurements, and telephone numbers. This collection of language-related user preferences are referred to as a *locale*. Designing your software to accommodate international audiences requires supporting these different formats.

Windows provides a standard means for inquiring what the default format is and also allows the user to change those properties. Your software can allow the user to change formats, but restrict these changes to your application or document type, rather than affecting the system defaults. Table 14.1 lists the most common format categories.

For more information about the functions that provide access to the current locale formats, see the documentation included in the Win32 SDK.

**Table 14.1 Formats for International Software**

| Category | Format considerations |
|---|---|
| Date | Order, separator, long or short formats, and leading zero |
| Time | Separator and cycle (12-hour vs. 24-hour), leading zero |
| Physical quantity | Metric vs. English measurement system |
| Currency | Symbol and format (for example, trailing vs. preceding symbol) |
| Separators | List, decimal, and thousandths separator |
| Telephone numbers | Separators for area codes and exchanges |
| Calendar | Calendar used and starting day of the week |
| Addresses | Order and postal code format |
| Paper sizes | U.S. vs. European paper and envelope sizes |

## Layout

For layout of controls or other elements in a window, it is important to consider alignment in addition to expansion of text labels. In Hebrew and Arabic countries, information is written right to left. So when localizing for these countries, reverse your U.S. presentation.

Some languages include diacritical marks that distinguish particular characters. Fonts associated with these characters can require additional spacing.

In addition, do not place information or controls into the title bar area. This is where Windows places special user controls for configurations that support multiple languages.

## References to Unsupported Features

Avoid confusing your international users by leaving in references to features that do not exist in their language version. Adapt the interface appropriately for features that do not apply. For example, some language versions may not include a grammar checker or support for bar codes on envelopes. Remove references to features such as menus, dialog boxes, and Help files from the installation program.

## Network Computing

Windows provides an environment that allows the user to communicate and share information across the network. When designing your software, consider the special needs that working in such an environment requires.

Conceptually, the network is an expansion of the user's local space. The interface for accessing objects from the network should not differ significantly from or be more complex than the user's desktop.

## Leverage System Support

When designing for network access, support standard conventions and interfaces, including the following:

- Use universal naming convention (UNC) paths to refer to objects stored in the file system. This convention provides transparent access to objects on the network.
- Use system-supported user identification that allows you to determine access without including your own password interface.
- Adjust window sizes and positions based on the local screen properties of the user.
- Avoid assuming the presence of a local hard disk. It is possible that some of your users work with diskless workstations.

## Client-Server Applications

Users operating on a network may wish to run your application from a network server. For applications that store no state information, no special support is required. However, if your application stores state information, design your application with a server set of components and a client set of components. The server components include the main executable files, dynamic link libraries, and any other files that need to be shared across the network. The client components consist of the components of the application that are specific to the user, including local registry information and local files that provide the user with access to the server components.

For information about installing the client and server components of your application, see Integrating with the System.

## Shared Data Files

When storing a file in the shared space of the network, it should be readily accessible to all users, so design the file to be opened multiple times. The granularity of concurrent access depends on the file type; that is, for some files you may only support concurrent access by word, paragraph, section, page, and so on. Provide clear visual cues as to what information can be changed and what cannot. Where multiple access is not easily supported, provide users with the option to open a copy of the file, if the original is already open.

## Record Processing

Record processing or transaction-based applications may require somewhat different structuring than the typical productivity application. For example, rather than opening and saving discrete files, the interface for such applications focuses on accessing and presenting data as records through multiple views, forms, and reports. One of the distinguishing and most important design aspects of record-processing applications is the definition of how the data records are structured. This dictates what information can be stored and in what format.

However, you can apply much of the information in this guide to record-oriented applications. For example, the basic principles of design and methodology are just as applicable as they are for individual file-oriented applications. You can also apply the guide's conventions for input, navigation, and layout when designing forms and report designs. Similarly, your can apply other secondary window conventions for data-entry design, including the following:

- Provide reasonable default values for fields.
- Use the appropriate controls. For example, use drop-down list boxes instead of long lists of option buttons.
- Distinguish text entry fields from read-only text fields.
- Design for logical and smooth user navigation. Order fields as the user needs to move through them. Auto-exit text boxes are often good for input of predefined data formats, such as time or currency inputs.
- Provide data validation as close to the site of data entry as possible. You can use input masks to restrict data to specific types or list box controls to restrict the range of input choices.

## Telephony

Windows provides support for creating applications with telephone communications, or *telephony*, services. Those services include the Assisted Telephony services for adding minimal, but useful telephonic functionality to applications, and the full Telephony API, for implementing full telephonic applications.

For more information about creating applications using the Microsoft Windows Telephony API (TAPI), see the documentation included in the Win32 SDK.

You should consider the following guidelines when developing telephony applications:

- Provide separate fields for users to enter country code, area code, and a local number. You may use auto-exit style navigation to facilitate the number entry. You can also use a drop-down list box to allow users to select a country code. The system provides support for listing available codes.
- Provide access to the TAPI Dialing Properties property sheet window wherever a user enters a phone number. This window provides a consistent and easy interface for users.
- Use the modem configuration interfaces provided by the system. If the user has not installed a modem, run the Windows TAPI modem installation wizard.

## Microsoft Exchange

Microsoft Exchange is the standard Windows interface for email, voice mail, FAX, and other communication media. Applications interact with Microsoft Exchange by using the Messaging API (MAPI) and support services and components.

For more information about MAPI, see the documentation included in the Win32 SDK.

Microsoft Exchange allows you to create support for an information service. An information service is a utility that enables messaging applications to send and receive messages and files, store items in an information store, obtain user addressing information, or any combination of these functions.

## Coexisting with Other Information Services

Microsoft Exchange is designed to simultaneously support different information services. Therefore when designing an information service, avoid:

- Initiating lengthy operations.
- Assuming exclusive use of key hardware resources, such as communications (COMM) ports and modems.
- Adding menu commands that are incompatible with other services.

## Adding Menu Items and Toolbar Buttons

Microsoft Exchange allows you to add menu items and toolbar buttons to the main viewer window. Follow the recommendations in this guide for defining menu and toolbar entries. In addition, where possible, define your menu items and toolbar entries (or their tooltips) in a way that allows the user to clearly associate the functionality with a specific information service.

## Supporting Connections

When the user selects an information service that you support, provide the user with a dialog box to confirm the choice and allow the user access to configuration properties. Because simultaneous services run at the same time, clearly identify the service. At the top of the window, display the icon and name of the service. You can include an option to not display the dialog box.

## Installing Information Services

Microsoft Exchange includes a special wizard for installation of information services. You can support this wizard to allow the user to easily install your service.

The system also provides profiles and files that define which services are available to users when they log on. When the user installs your service, ask the user which profile they would like to include with your service, such as their default profile.

## Introduction

The tables in this appendix summarize the basic mouse interface, including selection and direct manipulation (drag and drop).

## Interaction Guidelines for Common Unmodified Mouse Actions

**Table A. 1 Interaction Guidelines for Common Unmodified Mouse Actions**

| Action | Target | Effect on current selection state | Effect on anchor point location | Resulting operation using button 1 | Resulting op button 2 |
|---|---|---|---|---|---|
| Press | Unselected object | Clears the active selection. | Resets the anchor point to the object. | Selects the object. | Selects the o |
| | Selected object | None | None | None[1] | None |
| | White space (background) | Clears the active selection. | Resets the anchor point to the button down location. | Initiates a region (marquee) selection. | Initiates a reg selection. |
| Click | Unselected object | Clears the active selection. | Resets the anchor point to the object. | Selects the object. | Selects the o its pop-up me |
| | Selected object | None[2] | None[2] | Selects the object.[1] | Selects the o displays the s menu. |
| | White space (background) | Clears the active selection. | None | None | Displays the the white spa |
| Drag | Unselected object | Clears the active selection. | Resets the anchor point to the object. | Selects the object and carries out the default transfer operation[4] upon the button release at the destination. | Selects the o the non-defau menu[4] upon t at the destina |
| | Selected object | None | None | Carries out the default transfer operation[4] on the selection upon the button release at the destination. | Displays the transfer pop-u the button rel destination. |
| | White space (background) | Clears the active selection. | None | Selects everything logically included from anchor point to active end. | Selects every included from active end an menu for the selection. |
| Double-click | Unselected object | Clears the active selection. | Resets the anchor point to the object. | Selects the object and carries out the default operation. | Selects the o |
| | Selected object | None | None | Carries out the selection's default operation. | Selects the o |
| | White space (background) | Clears the active selection. | None | Carries out the default operation for the white space.[3] | None. |

1 Alternatively, you can support subselection for this action. Subselection means to distinguish an object in a selection for some purpose. For example, in a selection of objects, subselecting an object may define that object as the reference point for alignment commands.

2 Alternatively, you can support clearing the active selection and reset the anchor point to the object — if this better fits the context of the user's task.

3 The white space (or background) is an access point for commands of the view, the container, or both. For example, white space can include commands related to selection (Select All), magnification (Zoom), type of view (Outline), arrangement (Arrange By Date), display of specific view elements (Show Grid), general operation of the view (Refresh), and containment commands that insert objects (Paste).

4 The default transfer operation is determined by the destination of the drag and drop. Similarly, the destination determines the transfer commands displayed in the resulting pop-up menu when the mouse button is released. If the object cannot be dragged, then you can optionally use this action to create a range selection.

## Interaction Guidelines for Using the SHIFT Key to Modify Mouse Actions

**Table A. 2 Interaction Guidelines for Using the SHIFT Key to Modify Mouse Actions**

| Action | Target | Effect on current selection state | Effect on anchor point location | Resulting operation using button 1 | Resul butto |
|---|---|---|---|---|---|
| SHIFT+Press | Unselected object | Clears the active selection.1 | None | Extends the selection state from the anchor point to the object.2 | Exten from t object |
| | Selected object | Clears the active selection.1 | None | Extends the selection state from the anchor point to the object.2 | Exten from t object |
| | White space (background) | Clears the active selection.1 | None | Extends the selection state from the anchor point to the object logically included at the button down point.2 | Exten from t object buttor |
| SHIFT+Click | Unselected object | Clears the active selection.1 | None | Extends the selection state from the anchor point to the object.2 | Exten from t object up me select |
| | Selected object | Clears the active selection.1 | None | Extends the selection state from the anchor point to the object.2 | Exten from t object up me select |
| | White space (background) | Clears the active selection.1 | None | Extends the selection state from the anchor point to the object logically included at the button down point.2 | Exten from t object the bu displa the re |
| SHIFT+Drag | Unselected object | Clears the active selection.1 | None | Extends the selection state from the anchor point to the object.2 | Exten from t object up me select |
| | Selected object | Clears the active selection.1 | None | Extends the selection state from the anchor point to the object.2 | Exten from t object up me select |
| | White space (background) | Clears the active selection.1 | None | Extends the selection state from the anchor point to the object logically included at the button down point.2 | Exten from t object buttor displa the re |
| SHIFT+Double-click | Unselected object | Clears the active selection.1 | Resets the anchor point to the object. | Extends the selection state from the anchor point to the object2 and carries out the default command on the | Exten from t object |

| | | | | |
|---|---|---|---|---|
| | | | resulting selection.[3] | |
| Selected object | None | None | Extends the selection state from the anchor point to the object[2] and carries out the default command on the resulting selection.[3] | Exten from t object |
| White space (background) | Clears the active selection.[1] | None | Extends the selection state from the anchor point to the object logically included at the button down point[2] and carries out the default command on the resulting selection.[3] | Exten from t object buttor |

1 Only the active selection is cleared. The active selection is the selection made from the current anchor point. Other selections made by disjoint selection techniques are not affected, unless the new selection includes those selected elements.

2 The resulting selection state is based on the selection state of the object at the anchor point. If that object becomes selected, all the objects included in the range are selected. If the object is not selected, all the objects included in the range are also not selected.

3 If the effect of extending the selection unselects the object or a range of objects, the operation applies to the remaining selected objects.

## Interaction Guidelines for Using the CTRL Key to Modify Mouse Actions

**Table A. 3 Interaction Guidelines for Using the CTRL Key to Modify Mouse Actions**

| Action | Target | Effect on selection state | Effect on anchor point location | Resulting operation using button 1 | Resul butto |
|---|---|---|---|---|---|
| CTRL+Press | Unselected object | None | Resets the anchor point to the object. | Selects the object.[1] | Selec |
| | Selected object | None | Resets the anchor point to the object. | None | None |
| | White space (background) | None | Resets the anchor point to the button down location. | Initiates a disjoint region selection. | Initiat select |
| CTRL+Click | Unselected object | None | Resets the anchor point to the object. | Selects the object.[1] | Selec displa the en |
| | Selected object | None | Resets the anchor point to the object. | Unselects the object.[1] | Unsel displa the re |
| | White space (background) | None | None | None | Displa the ex |
| CTRL+Drag | Unselected object | None | Resets the anchor point to the object. | Selects the object[1] and copies the entire selection.[2] | Selec displa menu the de |
| | Selected object | None | Resets the anchor point to the object. | Copies the entire selection to the destination defined at the button up location.[2] | Selec displa menu the de |
| | White space (background) | None | None | Toggles the selection state of objects logically included by region selection.[3] | Toggle object region the po resulti |
| CTRL+Double-click | Unselected object | None | Resets the anchor point to the object. | Selects the object[1] and carries out the default command on the selection set. | Selec |
| | Selected object | None | Resets the anchor point to the object. | Unselects the object and carries out the default command on the selection set.[4] | Unsel |
| | White space (background) | None | None | Carries out the default command on the existing selection.[5] | None |

1 The CTRL key toggles the selection state of an object; this table entry shows the result.

2 If the user releases the CTRL key before releasing the mouse button, the operation reverts to the default transfer operation (as determined by the destination). If the destination does not support a copy operation, it may reinterpret operation. If the object cannot be dragged, you can optionally use this operation to create a disjoint range selection.

3 The range of objects included are all toggled to the same selection state, which is based on the first object included by the bounding region (marquee).

4 If the effect of toggling cancels the selection of the object, the operation applies to the remaining selected objects.

5 The white space (background) is an access point to the commands of the view, the container, or both.

# Introduction

This appendix summarizes the common keyboard operations, shortcut keys, and access key assignments.

## Common Navigation Keys

Table B.1 displays a summary of the keys used for navigation.

**Table B.1 Common Navigation Keys**

| Key | Cursor movement | CTRL+cursor movement |
|---|---|---|
| LEFT ARROW | Left one unit. | Left one proportionally larger unit. |
| RIGHT ARROW | Right one unit. | Right one proportionally larger unit. |
| UP ARROW | Up one unit or line. | Up one proportionally larger unit. |
| DOWN ARROW | Down one unit or line. | Down one proportionally larger unit. |
| HOME | To the beginning of the line. | To the beginning of the data (topmost position). |
| END | To the end of the line. | To the end of the data (bottommost position). |
| PAGE UP | Up one screen (previous screen, same position).[1] | Left one screen (or previous unit, if left is not meaningful). |
| PAGE DOWN | Down one screen (next screen, same position).[1] | Right one screen (or next unit, if right is not meaningful). |
| TAB[2] | Next field. | To next tab position (in property sheets, next page). |

1 “Screen” is defined as the height of the visible area being viewed. When scrolling, leave a nominal portion of the previous screen to provide context. For example in text, PAGE DOWN includes the last line of the previous screen as its first line.

2 Using the SHIFT key with the TAB key navigates in the reverse direction.

## Common Shortcut Keys

Table B.2 lists the common shortcut keys. Avoid assigning these keys to functions other than those listed.

**Table B.2 Common Shortcut Keys**

| Key | Meaning |
|---|---|
| CTRL+C (1) | Copy |
| CTRL+O | Open |
| CTRL+P | Print |
| CTRL+S | Save |
| CTRL+V (1) | Paste |
| CTRL+X (1) | Cut |
| CTRL+Z (1) | Undo |
| F1 | Display contextual Help window. |
| SHIFT+F1 | Activate context-sensitive Help mode (What's This?). |
| SHIFT+F10 | Display pop-up menu. |
| SPACEBAR (2) | Select (same as mouse button 1 click). |
| ESC | Cancel |
| ALT | Activate or inactivate menu bar mode. |
| ALT+TAB (3) | Display next primary window (or application). |
| ALT+ESC (3) | Display next window. |
| ALT+SPACEBAR | Display pop-up menu for the window. |
| ALT+HYPHEN | Display pop-up menu for the active child window (MDI). |
| ALT+ENTER | Display property sheet for current selection. |
| ALT+F4 | Close active window. |
| ALT+F6 (3) | Switch to next window within application (between modeless secondary windows and their primary window). |
| ALT+PRINT SCREEN | Capture active window image to the Clipboard. |
| PRINT SCREEN | Capture desktop image to the Clipboard. |
| CTRL+ESC | Access Start button in taskbar. |
| CTRL+F6 | Display next child window (MDI). |
| CTRL+TAB | Display next tabbed page or child window (MDI). |
| CTRL+ALT+DEL | Reserved for system use. |

1 The system still supports shortcut assignments available in earlier versions of Microsoft Windows (ALT+BACKSPACE, SHIFT+INSERT, CTRL+INSERT, SHIFT+DELETE). You should consider supporting them (though not documenting them) to support the transition of users.

2 If the context (for example, a text box) uses the SPACEBAR for entering a space character, you can use CTRL+SPACEBAR. If that is also defined by the context, define your own key.

3 Using the SHIFT key with this key combination navigates in the reverse direction.

## Windows Keys

Table B.3 lists shortcut key assignments for keyboards supporting the new Windows keys. The Left Windows key and Right Windows key are handled the same. All Windows key combinations, whether currently assigned or not, are strictly reserved for definition by the system only. Do not use this key for your own application-defined functions.

**Table B.3 Windows Keys**

| Key | Meaning |
|---|---|
| APPLICATION key | Display pop-up menu for the selected object. |
| WINDOWS key | Display Start button menu. |
| WINDOWS+F1 | Display Help Topics browser dialog box for the main Windows Help file. |
| WINDOWS+TAB | Activate next application window. |
| WINDOWS+E | Explore My Computer. |
| WINDOWS+F | Find a file. |
| WINDOWS+CTRL+F | Find a computer. |
| WINDOWS+M | Minimize All. |
| SHIFT+WINDOWS+M | Undo Minimize All. |
| WINDOWS+R | Display Run dialog box. |
| WINDOWS+BREAK | Reserved system function. |
| WINDOWS+*number* | Reserved for computer manufacturer use. |

## Accessibility Keys

Table B.4 lists the key combinations and sequences the system uses to support accessibility. Support for these options is set by users with the Windows Accessibility Options.

**Table B.4 Accessibility Keys**

| Key | Meaning |
|---|---|
| LEFT ALT+LEFT SHIFT+PRINT SCREEN | Toggle High Contrast mode |
| LEFT ALT+LEFT SHIFT+NUM LOCK | Toggle MouseKeys |
| SHIFT (pressed five consecutive times) | Toggle StickyKeys |
| RIGHT SHIFT (held eight or more seconds) | Toggle FilterKeys (SlowKeys, RepeatKeys, and BounceKeys) |
| NUM LOCK (held five or more seconds ) | Toggle ToggleKeys |

## Access Key Assignments

Table B.5 lists the recommended access key assignments for common commands. While the context of a command may affect specific assignments, you should use these access keys when you including these commands in your menus and command buttons.

**Table B.5 Access Key Assignments**

| | | |
|---|---|---|
| About | Insert Object | Quick View |
| Always on Top | Link Here | Redo |
| Apply | Maximize | Repeat |
| Back | Minimize | Restore |
| Browse | Move | Resume |
| Close | Move Here | Retry |
| Copy | New | Run |
| Copy Here | Next | Save |
| Create Shortcut | No | Save As |
| Create Shortcut Here | Open | Select All |
| Cut | Open With | Send To |
| Delete | Paste | Show |
| Edit | Paste Link | Size |
| Exit | Paste Shortcut | Split |
| Explore | Page Setup | Stop |
| File | Paste Special | Undo |
| Find | Pause | View |
| Help | Play | What's This? |
| Help Topics | Print | Window |
| Hide | Print Here | Yes |
| Insert | Properties | |

Avoid assigning access keys to OK and Cancel when the ENTER key and ESC key, respectively, are assigned to them by default.

## Introduction

The following checklist summarizes the guidelines covered in this guide. You can use this guideline summary to assist you in your planning, design, and development process.

Remember, the objective of the recommendations and suggestions in this guide is to benefit your users, not to enforce a rigid set of rules. Consistency in design makes it easier for a user to transfer skills from one task to another. When you need to diverge from or extend these guidelines, follow the principles and spirit of this guide.

## General Design

- Supports user initiation of actions
- Supports user customization of the interface
- Supports an interactive and modeless environment
- Supports direct manipulation interfaces
- Uses familiar, appropriate metaphors
- Is internally consistent; similar actions have a similar interface
- Makes actions reversible where possible; where not possible, requests confirmation
- Makes error recovery easy
- Eliminates possibilities for user errors, where possible
- Uses visual cues to indicate user interaction
- Provides prompt feedback
- Provides feedback that is appropriate to the task
- Makes appropriate use of progressive disclosure

## Design Process

- Employs a balanced team
- Uses an iterative design cycle
- Incorporates usability assessment as a part of the process
- Designs for user limitations

## Input and Interaction

- Follows basic mouse interaction guidelines
- Uses appropriate modifier keys for adjusting or adding elements to a selection
- Uses appropriate visual feedback, such as highlighting or handles, to indicate selected objects
- Supports default and nondefault drag and drop
- Supports standard transfer commands, where appropriate
- Provides keyboard interface for all basic operations
- Follows keyboard guidelines for navigation, shortcut keys, and access keys
- Keeps foreground activity as modeless as possible
- Indicates use of modes visually
- Provides access to common, basic operations through single click interaction
- Provides shortcut methods (such as double-clicking) to common or frequently used operations for experienced users

## Windows

- Provides title text for all windows and follows guidelines for defining correct title bar text and icon
- Supports single window instance model: brings the existing window to the top of the Z order when the user attempts to reopen a view or window that is already open
- Uses common dialog boxes, where applicable
- Follows common dialog box conventions when substituting these dialog boxes
- Saves and restores the window state
- Adjusts window size and position to the appropriate screen size
- Uses modeless secondary windows, wherever possible
- Avoids system modal secondary windows, except in the case of possible loss of data
- Automatically supplies a proposed name upon the creation of a new object
- Uses the appropriate message symbol in message boxes
- Provides a brief but clear statement of problem and possible remedies in message boxes
- Organizes properties into property sheets, using property pages for peer properties and list controls for hierarchical navigation
- Places command buttons that apply to the page inside a tabbed page (for example, a property sheet), and outside of a page when the user applies by window (as a set)
- Follows single document window interface (SDI) or multiple document interface (MDI or MDI alternatives) conventions

## Control Usage

- Uses system-supplied controls, wherever possible
- Provides an object pop-up menu for the title bar icon
- Provides a pop-up menu for the window
- Avoids multiple level hierarchical interfaces (menus, secondary windows) for frequently used access operations
- Uses an ellipsis only for commands that require additional input or parameters
- Uses the menu (triangular) arrow image to indicate when a control can display more information (cascading menus, drop-down control arrows, scroll bar arrows)
- Provides pop-up menus for selections and other user identifiable objects
- Supports the display of pop-up menus using mouse button 2, the keyboard shortcut keys, and action handles
- Displays pop-up menus upon the release of the mouse button
- Follows guidelines for ordering the commands on pop-up menus
- Limits commands on pop-up menus to those that apply to the selection and its immediate context
- Makes toolbars user configurable (display, position, content)
- Uses defined toolbar label images when supporting common actions
- Defines custom controls to be visually and operationally consistent with standard system controls

## Integration

- Makes full and correct use of the registry, including registration of file extensions, file types, and icons
- Avoids use of Autoexec.bat, Config.sys, or initialization (.INI) files
- Supports and registers entries for Print and Print To interfaces for file types that are printable
- Provides and registers icons in 32-x 32-, 16-x 16-, and 48-x 48- pixel sizes for application, and all document and data file types (in both color and monochrome versions)
- Registers file types supported under the system's New command
- Uses system interfaces when adding property pages for types
- Supports long filenames and universal naming convention (UNC) paths, where files are used
- Displays filenames correctly
- Follows appropriate conventions when using the taskbar to support notification and status information
- Supports appropriate behavior for creating and integrating scrap objects
- Follows guidelines for installation
- Provides an uninstall program
- Provides appropriate support for network installation
- Supports all OLE user interface guidelines, including transfer interfaces (drag and drop and nondefault drag and drop), pop-up menus, and property sheets for OLE embedded and linked objects

## User Assistance

- Provides context-sensitive Help information for elements (including controls)
- Provides task Help topics for basic procedures
- Provides tooltips for all unlabeled controls, such as in toolbars
- Follows guidelines for messages, status bar information, contextual Help, task Help, online Reference Help, and wizards

## Visual Design

- Uses color only as an enhancing, secondary form of information
- Uses a limited set of colors
- Uses system metrics for all display elements (such as color settings and fonts)
- Uses standard border styles
- Uses appropriate appearance for visual states of controls
- Supports dimensionality using light source from the upper left
- Supports guidelines for design and appearance of controls and icons
- Supports guidelines for layout and font use
- Uses correct capitalization for control labels

## Sound

- Uses audio only for secondary cues (applicable only where audio is not the primary form of information, for example, music)
- Supports system interface for sound volume
- Supports and provides appropriate visual output for system ShowSounds setting

## Accessibility

- Clearly labels all controls, icons, windows, and other screen elements (even if not visible) so they can be identified by screen review and voice input utilities
- Indicates keyboard focus
- Uses standard functions for displaying text
- Makes components of graphic images that must be separately discernible by using metafiles, drawing each component separately, or by redrawing components with null operation (NOP) when the user has installed a screen review utility
- Avoids time-out interaction or makes timing interaction user configurable
- Avoids triggering actions on user navigation in the interface
- Supports scaling or magnification views where possible and applicable
- Supports system accessibility settings (such as High Contrast Mode) and appropriately adjusts the user interface elements
- Tests for compatibility with common accessibility aids
- Includes people with disabilities in testing process
- Provides documentation in nonprinted formats, such as online
- Provides telephone support to users using text telephones (TT/TDD)

## International Users

- Provides sufficient space for character expansion for localization
- Avoids jargon and culturally dependent words or ideas
- Avoids using punctuation keys in shortcut key combinations
- Supports displaying information based on local formats
- Uses layout conventions appropriate to reading conventions
- Adjusts references to unsupported features

## Network Users

- Supports system naming and identification conventions
- Supports shared access for application and data files

# Introduction

This guide is primarily intended for applications designed for Microsoft Windows 95 and later releases. However, you can apply many of the conventions to other releases of Windows. This appendix covers the differences you may need to consider.

## Microsoft Windows 3.1

*The Windows Interface: An Application Design Guide* provided guidelines for applications designed for Microsoft Windows 3.1. It was included in the Microsoft Windows 3.1 Software Development Kit (SDK) and published by Microsoft Press®.

Many of the recommendations in *The Windows Interface: An Application Design Guide* were carried forward and extended into this guide to reflect the new conventions in Microsoft OLE and Microsoft Windows 95. These extended, revised, or new conventions include:

- Recommendations for applying command and direct manipulation transfer methods between applications and the system's shell components.
- Recommendations for mouse button 2, specifically, displaying pop-up menus upon a button 2 click and supporting nondefault drag and drop.
- The replacement of the Control (System) menu with the pop-up menu for the window.
- New conventions for minimizing and re-opening windows.
- Recommendations for using the title bar Close button, the What's This? button, and title bar icons and their accompanying pop-up menus.
- New guidelines for ordering the title bars of document or data file windows.
- New common dialog box interfaces and new controls — list views, tree views, column headings, progress indicators, toolbars, tooltips, property sheets, tabs, status bars, rich-text boxes, sliders, spin boxes, proportional scroll bars, and pen controls.
- Recommendations for displaying and editing properties; including guidelines for using the Properties command, property sheets, and property inspectors.
- New conventions for context-sensitive (What's This?) Help and task Help and recommendations for wizard design.
- New registry entries and shell integration conventions — support for storing application state and path information, file type association, file creation, adding commands for files, file installation, providing access to your application, extending the shell, file viewing using the Quick View command, adding sound events, icon support, and AutoPlay.
- Support for long filenames and access to network resources using universal naming conventions (UNC) pathnames.
- New OLE recommendations — container supplied pop-up menus, Properties command, and property sheets.
- Revised design conventions for window components and icons.
- Recommended conventions for supporting Microsoft telephony application programming interfaces (TAPI), messaging application programming interfaces (MAPI), Plug and Play, pen application programming interfaces, and accessibility utilities.

## Microsoft Windows NT 3.51

Windows NT 3.51 (and Windows NT Server 3.51) includes a special dynamic-link library (COMCTL32.DLL) that supports the new controls in Windows 95. As a result, you can develop applications for Windows 95 and Windows NT that have general functional and operational compatibility. However, when applying the conventions in this guide to applications designed for Windows NT, be aware of the following differences in release 3.51:

- Window visuals and shell components follow the Windows 3.1 appearance and operation.
- Close buttons and title bar icons are not supported.
- Open and Save As common dialog boxes follow Windows 3.1 appearance and conventions.
- Message box symbols follow Windows 3.1 conventions.
- Pen API interfaces are not supported.
- Registry formats and entries — support for application state and path information, shell creation, the Quick View command, adding commands for files, shell extensions, and sound event registration.
- Program Manager is still the primary interface for providing user access to applications. File Manager, rather than Windows Explorer, supports file browsing and file management.
- There is no support for the Add/Remove Programs installation object (included in Control Panel). Instead, provide an object in your application's Program Manager group.
- AutoPlay is not supported.
- Taskbar and desktop toolbars are not supported.
- The Recycle Bin is not supported.
- The Passwords object (in Control Panel) is not supported. Use the Windows NT User Manager instead.
- Microsoft MAPI 1.0, TAPI, and Plug and Play are not supported. (Simple MAPI support is included.)
- Some system shortcut key assignments, such as CTRL+ESC and CTRL+ALT+DEL, operate differently.

For more information about these interfaces, see the documentation included in the Microsoft Win32 Software Development Kit (SDK).

## General Design

Baecker, Ronald M., and Buxton, William A. S. *Readings in Human-Computer Interaction: A Multidisciplinary Approach*. Los Altos, Calif.: M. Kaufmann, 1987.

Brooks, Frederick P. *The Mythical Man-Month: Essays on Software Engineering*. Reading, Mass.: Addison-Wesley Pub. Co., 1975.

Heckel, Paul. *The Elements of Friendly Software Design.* New Ed., San Francisco: SYBEX, 1991.

Lakoff, George, and Johnson, Mark. *Metaphors We Live By.* Chicago: University of Chicago Press, 1980.

Laurel, Brenda, Ed. *The Art of Human-Computer Interface Design*. Reading, Mass.: Addison-Wesley Pub. Co., 1990.

Norman, Donald A. *The Design of Everyday Things*. New York: Basic Books, 1990.

Norman, Donald A., and Draper, Stephen, W., Eds. *User Centered System Design*: *New Perspectives on Human-Computer Interaction*. Hillsdale, N.J.: L. Erlbaum Associates, 1986.

Shneiderman, Ben. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Reading, Mass.: Addison-Wesley, 1992.

Tognazzini, Bruce. *Tog on Interface*. Reading, Mass.: Addison-Wesley, 1992.

## Graphic Information Design

Blair, Preston. *Cartoon Animation*. How to Draw and Paint Series. Tustin, Calif.: Walter Foster Pub., 1989.

Dreyfuss, Henry. *Symbol Sourcebook: An Authoritative Guide to International Graphic Symbols*. New York: Van Nostrand Reinhold Co., 1984.

Thomas, Frank., and Johnston, Ollie. *Disney Animation: The Illusion of Life*. New York: Abbeville Press, 1984.

Tufte, Edward R. *Envisioning Information*. Cheshire, Conn.: Graphics Press, 1990.

Tufte, Edward R. *The Visual Display of Quantitative Information*. Cheshire, Conn.: Graphics Press, 1983.

## Usability

Dumas, Joseph S., and Redish, Janice C. *A Practical Guide to Usability Testing.* Norwood, N.J.: Ablex Pub. Corp., 1993.

Nielsen, Jakob. *Usability Engineering*. Boston: Academic Press, 1993.

Rubin, Jeffrey. *Handbook of Usability Testing: How to Plan, Design, and Conduct Effective Tests*. New York: Wiley, 1994.

Whiteside, John, Bennett, John, and Holtzblatt, Karen. "Usability Engineering: Our Experience and Evolution." In *Handbook of Human-Computer Interaction*, Martin Helander (Ed.), Elsevier Science Pub. Co., Amsterdam, 1988.

Wiklund, Michael E., Ed. *Usability in Practice: How Companies Develop User-Friendly Products*. Boston: AP Professional, 1994.

## Object-Oriented Design

Booch, Grady. *Object-Oriented Analysis and Design with Applications*. Redwood City, Calif.: Benjamin/Cummings Pub. Co., 1994.

Peterson, Gerald E., Ed. *Tutorial: Object-Oriented Computing: Volume 2: Implementations*. Washington, D.C.: Computer Society Press of the IEEE, 1987.

Rumbaugh, James, et al. *Object-Oriented Modeling and Design*. Englewood Cliffs, N.J.: Prentice Hall, 1991.

## Accessibility

For a list of accessibility aids available for Microsoft Windows, accessibility software vendors, potential test sites, or facilities for producing accessible documentation, contact:

Microsoft Sales Information Center
One Microsoft Way
Redmond, WA 98052-6399
(800) 426-9400 (voice)
(800) 892-5234 (text telephone)
(206) 936-7329 (FAX)

An assistive technology program in your area can provide referrals to programs and services available to you. To locate the assistive technology program nearest to your location, contact:

National Information System
Center for Development Disabilities
University of South Carolina
Benson Building
Columbia, SC 29208
(803) 777-4435 (voice or text telephone)
(803) 777-6058 (FAX)

The Trace Research and Development Center publishes references and materials on accessibility, including:

Vanderheiden, Gregg C., and Vanderheiden, Katherine R. *Accessible Design of Consumer Products: Guidelines for the Design of Consumer Products to Increase Their Accessibility to People with Disabilities or Who Are Aging*. Madison, Wis.: Trace Research and Development Center, 1991.

Vanderheiden, Gregg C. *Application Software Design Guidelines: Increasing the Accessibility of Applications Software to People with Disabilities and Older Users (Version 1.1)*. Madison, Wis.: Trace Research and Development Center, 1994.

Borden, Fatherly, Ford, and Vanderheiden, Eds. *Trace Resource Book: Assistive Technologies for Communication, Control and Computer Access*. Madison, Wis.: Trace Research and Development Center, 1993.

For information on these books and other resources available from the Trace Research and Development Center, contact them at:

Trace Research and Development Center
University of Wisconsin - Madison
S-151 Waisman Center
1500 Highland Avenue
Madison, WI 5705-2280
(608) 263-2309 (voice)
(608) 263-5408 (text telephone)
(608) 262-8848 (FAX)

## Organizations

The following organizations publish journals and sponsor conferences on topics related to user interface design.

SIGCHI (Special Interest Group in Computer Human Interaction)
Association for Computing Machinery
1515 Broadway
New York, NY 10036-5701
212-869-7440

SIGGRAPH (Special Interest Group on Graphics)
Association for Computing Machinery
1515 Broadway
New York, NY 10036-5701
212-869-7440

Human Factors and Ergonomics Society
P.O. Box 1369
Santa Monica, CA 90406-1369
310-394-1811